RWTH Aachen

Department of Computer Science
Technical Report

Kolloquium Programmiersprachen und Grundlagen der Programmierung

Klaus Indermark und Thomas Noll (Hrsg.)

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2001-11

RWTH Aachen · Department of Computer Science · December 2001

Vorwort

Das Kolloquium über *Programmiersprachen und Grundlagen der Programmierung* steht in der Tradition einer Reihe von Arbeitstagungen, die ursprünglich von den Forschungsgruppen unter der Leitung von F.L. Bauer (TU München), K. Indermark (RWTH Aachen) und H. Langmaack (CAU Kiel) initiiert wurde und inzwischen ein breites Interesse gefunden hat. Anfang Oktober 2001 fand es zum elften Mal statt und wurde vom Lehrstuhl für Informatik II der RWTH Aachen ausgerichtet. Veranstaltungsort war das Hotel *Paulushof* in Simmerath–Rurberg am Rursee in der Eifel.

Die folgende Liste gibt eine Übersicht der vorherigen Tagungsorte sowie der jeweiligen Veranstalter:

1980: Tannenfelde bei Neumünster (Universität Kiel)
1982: Altenahr (RWTH Aachen)
1985: Passau (Universität Passau)
1987: Midlum auf Föhr (Universität Kiel)
1989: Hirschegg im Kleinwalsertal (Universität Augsburg)
1992: Rothenberge bei Steinfurt (Universität Münster)
1993: Barbarahütte auf der Kreuzeckalm, bei Garmisch-Partenkirchen (Universität der Bundeswehr München)
1995: Alt–Reichenau im Bayerischen Wald (Universität Passau)
1997: Avendorf auf Fehmarn (Universität Kiel)
1999: Kirchhundem–Heinsberg im Rothaargebirge (FernUniversität Hagen)

In diesem Jahr trafen sich 44 Informatikerinnen und Informatiker von den Universitäten Augsburg, Dortmund, Freiburg, Kiel, Köln, Münster, Ulm, der RWTH Aachen, den Technischen Universitäten Berlin, Dresden und München, der FernUniversität Hagen, der Medizinischen Universität Lübeck und der Universität der Bundeswehr München. Der vorliegende Bericht enthält die Vortragsausarbeitungen der Teilnehmer.

Der herzliche Dank der Organisatoren gebührt allen Vortragenden und Teilnehmern für ihre Beiträge in Wort und Schrift. Frau Elke Ohlenforst und Herrn Arnd Gehrmann sowie den übrigen Mitarbeitern des Lehrstuhls für Informatik II danken wir für die Mithilfe bei der Planung und Durchführung des Kolloquiums.

Während der Veranstaltung hat sich dankenswerterweise Herr Prof. Peter Thiemann von der Universität Freiburg bereiterklärt, das nächste Kolloquium im Jahr 2003 auszurichten.

Aachen,
Dezember 2001

Klaus Indermark
Thomas Noll

Inhaltsverzeichnis

Programmverifikation

Semantik von Programmiersprachen

Anwendungssysteme

Untersuchungen von Algorithmen für transitive Reduktionen und minimale Äquivalenzgraphen

Rudolf Berghammer, Thorsten Hoffmann und Christian Kasper

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Olshausenstraße 40, D-24098 Kiel

1 Einleitung

Viele Problemstellungen der Informatik lassen sich abstrahieren und mit Mitteln anderer Disziplinen wie beispielsweise der Graphentheorie lösen. So können z.B. das Finden von Abhängigkeiten in Datenbanken oder Inkonsistenzen in Ablaufplänen durch geeignete Abstraktion auf einfache Weise als Erreichbarkeitsprobleme für gerichteten Graphen formuliert werden. Zu einem gerichteten Graphen $g = (V, R)$ mit Knotenmenge V und Pfeilrelation R kann diese Erreichbarkeitsinformation an der reflexiv-transitiven Hülle R^* abgelesen werden. Im folgenden werden wir viele graphentheoretische Sachverhalte mit Hilfe von relationenalgebraischen Formeln beschreiben. Wir gehen daher davon aus, daß der Leser mit den grundlegenden Operationen auf Relationen, wie Vereinigung ($R \cup S$), Durchschnitt ($R \cap S$), Negation ($\overline{R}$), Transposition (R^T) und Produkt (RS) vertraut ist. Eine gute Einführung in dieses Themengebiet bietet das Buch [4]. Unser Ziel besteht nun darin, einen Teilgraphen $h = (V, S)$ von g zu finden, dessen Pfeilrelation S möglichst wenig Einträge enthält und dessen reflexiv-transitive Hülle mit der von g übereinstimmt. Auf diese Weise enthält h dieselbe Erreichbarkeitsinformation wie g, wobei jedoch weniger Platz zur Speicherung von h benötigt wird. Einen Teilgraphen $h = (V, S)$ von g mit anzahlminimaler Pfeilrelation S, so daß $R^* = S^*$ gilt, bezeichnet man als *minimalen Äquivalenzgraphen* von g. Da die Berechnung eines solchen minimalen Äquivalenzgraphen ein NP-hartes Problem ist, können wir nicht erwarten, einen effizienten Algorithmus für die Lösung dieses Problems zu finden. Stattdessen beschäftigen wir uns mit der Approximation von minimalen Äquivalenzgraphen, indem wir Teilgraphen $h = (V, S)$ von g betrachten, so daß S eine inklusionsminimale Teilmenge von R mit der Eigenschaft $R^* = S^*$ ist. Solche Teilgraphen bezeichnen wir als *transitive Reduktionen* von g. Diese sind zwar im allgemeinen nicht eindeutig und auch nicht anzahlminimal, lassen sich dafür aber effizient berechnen, was wir im folgenden zeigen werden. Wir stellen zunächst in Abschnitt 2 ein generisches Minimierungsverfahren vor, das wir in Abschnitt 3 durch geeignete Instantiierung zur Berechnung von transitiven Reduktionen stark zusammenhängender Graphen verwenden. Die Einschränkung auf diese Art von Graphen ist dadurch gerechtfertigt, daß sich eine transitive Reduktion eines beliebigen Graphen aus den transitiven Reduktionen der starken Zusammenhangskomponenten und des kreisfreien reduzierten Graphen, in dem jeder Knoten einer starken Zusammenhangskomponente entspricht, zusammensetzen läßt, was in [3] gezeigt wurde.

2 Ein generisches Minimierungsverfahren

Die Berechnung einer transitiven Reduktion stellt einen Spezialfall eines allgemeinen Minimierungsproblems dar, nämlich zu einer Menge M, einem Prädikat $\mathcal{P}$ auf der Potenzmenge 2^M und einem Element R dieser Potenzmenge eine inklusionsminimale Teilmenge von R zu berechnen, die $\mathcal{P}$ erfüllt. Nachfolgend geben wir einen Algorithmus zur Lösung dieses allgemeineren Problems an, wobei wir voraussetzen, daß das Prädikat $\mathcal{P}$ *nach oben vererbend* ist, d.h. für alle $X, Y \in 2^M$ gilt: Aus $X \subseteq Y$ und $\mathcal{P}(X)$ folgt $\mathcal{P}(Y)$. Die Nachbedingung kann nun wie folgt formuliert werden, wobei wir die Variable A dazu verwenden, das Resultat abzuspeichern:

$$post(R, A) \iff \mathcal{P}(A) \ \wedge \ A \subseteq R \ \wedge \ \forall X \in 2^A : \mathcal{P}(X) \Rightarrow X = A$$

Durch Generalisierung der Nachbedingung und Einführung einer neuen Variable B gelangt man zu folgender Invariante:

$$inv(R, A, B) \iff \mathcal{P}(A) \ \wedge \ A \subseteq R \ \wedge \ B \subseteq A \ \wedge \ \forall x \in A \setminus B : \neg\mathcal{P}(A \setminus \{x\})$$

Setzen wir als Vorbedingung voraus, daß R das Prädikat $\mathcal{P}$ erfüllt, so erhält man durch Programmentwicklung mit Hilfe der Invariantentechnik nachfolgendes while-Programm, wobei durch die Operation $elem(B)$ ein Element aus B ausgewählt wird:

```
Minimum(R)
   DECL S, A, B, b
   BEG  A, B := S, S;
        WHILE B ≠ ∅ DO
          b := elem(B);
          IF P(A \ {b}) THEN A, B := A \ {b}, B \ {b}
                        ELSE B := B \ {b} FI OD
      RETURN A
   END.
```

Dieses Programm verwenden wir im nächsten Abschnitt als Grundgerüst, um transitive Reduktionen von stark zusammenhängenden Graphen zu berechnen.

3 Berechnung transitiver Reduktionen

Ziel dieses Abschnitts ist es, ein effizientes Programm zur Berechnung einer transitiven Reduktion stark zusammenhängender Graphen vorzustellen. Dafür setzen wir für den Rest des Abschnitts einen stark zusammenhängender Graphen $g = (V, R)$ voraus, wobei der starke Zusammenhang durch die Formel $R^* = \mathsf{L}$ beschrieben wird. Das Prädikat $\mathcal{P}$ definieren wir nun auf der Menge aller homogenen Relationen auf V durch $\mathcal{P}(X) \stackrel{\wedge}{=} X^* = \mathsf{L}$. Dieses Prädikat ist trivialerweise nach oben vererbend, so daß das Programm *Minimum* aus dem letzten Abschnitt mit dieser Instantiierung eine transitive Reduktion von g berechnet. Die Laufzeit des Programms hängt im wesentlichen von zwei Faktoren ab. Zum einen von der Anzahl der Schleifendurchläufe und zum anderen von der Effizienz der Prädikatauswertung innerhalb der Schleife. Um die Anzahl der Schleifendurchläufe zu verringern, erscheint es sinnvoll, eine Vorberechnung durchzuführen, so daß A und B mit einer Relation S initialisiert werden, die in R enthalten ist, ebenfalls $S^* = \mathsf{L}$ erfüllt und weniger Einträge als R enthält.

Zu diesem Zweck führen wir den Begriff des gerichteten Baumes ein. Gegeben seien eine homogene Relation T und ein Punkt r. Dann heißt T ein *gerichteter Baum mit Wurzel* r, falls $r\mathsf{L} \subseteq T^*$, $TT^\mathsf{T} \subseteq \mathsf{I}$ und $T^+ \subseteq \overline{\mathsf{I}}$ gilt. Das nachfolgende Lemma zeigt, wie man nun im Fall $R^* = \mathsf{L}$ aus zwei in R bzw. R^T enthaltenen gerichteten Bäumen mit gleicher Wurzel eine Relation S gewinnen kann, die in R enthalten ist und $S^* = \mathsf{L}$ erfüllt.

Lemma 1. *Sind T_1 und T_2 zwei gerichtete Bäume mit gleicher Wurzel r und gelten $T_1 \subseteq R$ und $T_2 \subseteq R^\mathsf{T}$, so gelten auch $T_1 \cup {T_2}^\mathsf{T} \subseteq R$ und $(T_1 \cup {T_2}^\mathsf{T})^* = \mathsf{L}$.*

Initialisieren wir A und B mit einem solchen S so verringert sich die Anzahl der Schleifendurchläufe von maximal $|V|^2$ auf höchstens $2 \cdot |V| - 2$.

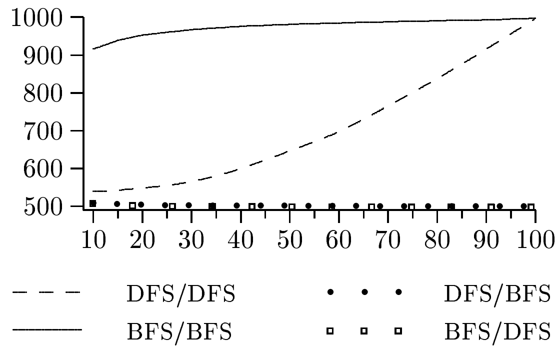
Eine weitere Laufzeitverbesserung kann erreicht werden, indem wir das Prädikat $\mathcal{P}$ durch ein anderes Prädikat $\mathcal{Q}$ ersetzen, das effizienter ausgewertet werden kann. Wir fordern für $\mathcal{Q}$ die Äquivalenz $\mathcal{P}(A \setminus \{b\}) \iff \mathcal{P}(A) \wedge \mathcal{Q}(A, b)$. Setzen wir nun $Q(A, b) \triangleq b \subseteq (A \cap \overline{b})^*$, so lautet die obige Äquivalenz $(A \cap \overline{b})^* = \mathsf{L} \iff A^* = \mathsf{L} \wedge b \subseteq (A \cap \overline{b})^*$. Da $A^* = \mathsf{L}$ laut Invariante gilt, können wir folglich in dem Konditional $\mathcal{P}(A \setminus \{b\})$ durch $\mathcal{Q}(A, b)$ ersetzen. Auf diese Weise wird der Test auf starken Zusammenhang durch einen Erreichbarkeitstest ersetzt, wodurch der Aufwand für die Prädikatauswertung von $\mathcal{O}(|V|^3)$ auf $\mathcal{O}(|V|)$ sinkt.

Insgesamt erhalten wir durch diese Modifikationen das nachfolgende relationale Programm:

```
TransRed(R)
   DECL S, A, B, b, r
   BEG  r := point(L);
        S := Tree1(R, r) ∪ Tree2(Rᵀ, r)ᵀ;
        A, B := S, S;
        WHILE B ≠ O DO
          b := atom(B);
          IF b ⊆ (A ∩ b̄)* THEN A, B := A ∩ b̄, B ∩ b̄
                           ELSE B := B ∩ b̄ FI OD
        RETURN A
   END.
```

Die Operationen point bzw. atom wählen aus einem nicht leeren Vektor bzw. einer nicht leeren Relation einen beliebigen Eintrag aus, und die Programme $Tree_1$ und $Tree_2$ dienen der Berechnung der Bäume T_1 und T_2 aus Lemma 1. Wählen wir für diese Programme Standardalgorithmen wie Breiten- oder Tiefensuche, so beträgt der Aufwand für die Vorberechnung $\mathcal{O}(|V|^2)$. Durch diese Vorberechnung wird die Schleife weniger als $|V|$ mal durchlaufen, und die Prädikatauswertung ist ebenfalls in Laufzeit $\mathcal{O}(|V|)$ möglich, so daß sich dieses Programm in einer Programmiersprache wie C oder Pascal insgesamt in Laufzeit $\mathcal{O}(|V|^2)$ realisieren läßt.

Interessant für die Praxis sind transitive Reduktionen mit möglichst wenig Pfeilen, da sie platzsparend gespeichert werden können und dieselben Erreichbarkeitsinformationen beinhalten wie der ursprüngliche Graph. Aufgrund der Vorberechnung enthält eine mit dem obigen Programm berechnete transitive Reduktion höchstens $2 \cdot |V| - 2$ Pfeile und mindestens $|V|$ Pfeile, was dann genau einem Hamiltonschen Kreis entspricht. Durch Verwendung verschiedener Baumalgorithmen für $Tree_1$ und $Tree_2$ gelangt man zu sehr unterschiedlichen Ergebnissen, wie die nachfolgende Grafik zeigt. Dabei haben wir zufällig erzeugte Graphen mit 500 Knoten untersucht und die Pfeildichte variiert. Für die Baumalgorithmen haben wir Breiten- und Tiefensuche in den vier möglichen Kombinationen verwendet, wobei folgendes Diagramm entstanden ist, bei dem die Pfeildichte auf der x-Achse aufgetragen ist, und die Anzahl der Knoten den Markierungen der y-Achse entspricht:



Dieses Ergebnis mag auf den ersten Blick überraschend erscheinen, läßt sich aber leicht erklären. Durch die Vorberechnung mit den beiden Baumalgorithmen erhalten wir zunächst durch $\mathit{Tree}_1(R,r)$ Pfade von der Wurzel r zu jedem übrigen Knoten von g. Mit Hilfe von $\mathit{Tree}_2(R^\mathsf{T},r)^\mathsf{T}$ werden Pfade von jedem von r verschiedenen Knoten zu r berechnet. Insgesamt besteht S folglich aus einer Vereinigung von Kreisen. Verwenden wir für die Vorberechnung zweimal die Breitensuche, so besteht S aus vielen kleinen Kreisen, bei denen mit zunehmender Dichte des Graphen g weniger Pfeile aus S entfernt werden können, ohne den starken Zusammenhang zu verlieren, was zu dem im Diagramm sichtbaren schlechten Ergebnis führt. Durch zweimaliges Anwenden der Tiefensuche gelangt man bei geringer Dichte noch zu großen Kreisen und somit zu recht guten Ergebnissen, aber bei zunehmender Dichte besitzen T_1 und T_2 mehr gemeinsame Knoten, was die Kreise wiederum verkleinert und so das Ergebnis verschlechtert. Sehr gute Resultate können mit einer Kombination aus Breiten- und Tiefensuche erzielt werden, unabhängig von der Dichte des Graphen. Dies liegt darin begründet, daß durch die Tiefensuche lange Pfade von der Wurzel r zu jedem anderen Knoten von g berechnet werden, während durch die Breitensuche von diesen Knoten aus der Kreis zur Wurzel mit sehr wenig Pfeilen geschlossen wird. Insgesamt lassen sich auf diese Weise in der Praxis Ergebnisse berechnen, die sehr nahe am Optimum liegen, auch wenn dieses Programm nur eine theoretische Güte von 2 hat, d.h. wenn eine optimale Lösung (ein minimaler Äquivalenzgraph) mit n Pfeilen exisiert, können wir eine Approximation berechnen, die maximal $2 \cdot n - 2$ Pfeile enthält.

In der Literatur (vgl. [2]) findet man einen weiteren Algorithmus zur Approximation eines minimalen Äquivalenzgraphen, der in nahezu linearer Laufzeit, bezogen auf die Pfeilanzahl, eine solche Approximation mit Güte 1,75 berechnet. Allerdings haben praktische Tests gezeigt, daß die berechneten Ergebnisse im allgemeinen keine transitiven Reduktionen sind und anderthalb mal mehr Pfeile enthalten als eine optimale Lösung. Um nun die bessere theoretische Güte dieses Verfahrens und die guten praktischen Resultate von unserem Algorithmus auszunutzen, bietet sich eine Kombination beider Verfahren an. Verwenden wir diesen Algorithmus für die Vorberechnung in unserem Programm, so verbessern wir die Güte von 2 auf 1,75 und behalten gleichzeitig die guten praktischen Ergebnisse bei.

4 Zusammenfassung

In diesem Artikel haben wir ein generisches Programm zur Berechnung inklusionsminimaler Teilmengen vorgestellt. Durch geeignete Instantiierung kann dieses Programm dazu verwendet werden, auf einfache Weise transitive Reduktionen stark zusammenhängender Graphen zu berechnen. Zu diesem Zweck werden in einer Vorberechungsphase zwei Baumalgorithmen verwendet, bei der die Kombination aus Tiefen- und Breitensuche zu sehr guten praktischen Ergebnissen führt, die im Durchschnitt nur um wenige Pfeile von einer optimalen Lösung abweichen. Allerdings ist die theoretische Güte des Verfahrens 2, kann aber durch eine neue Vorberechnungsphase, in der ein Programm (siehe [2]) zur Approximation eines minimalen Äquivalenzgraphen verwendet wird, auf 1,75 verringert werden.

Literatur

1. C. Kasper. Untersuchungen von Algorithmen für transitive Reduktionen und minimale Äquivalenzgraphen. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel (2001).
2. S. Khuller, B. Raghavachari, N. Young. Approximating the minimum equivalent digraph. SIAM Journal on Computing 24(4) (1995).
3. H. Noltemeier. Reduktion von Präzedenzstrukturen. Zeitschrift für Operations Research 20. Physica-Verlag (1976).
4. G. Schmidt, T. Ströhlein. Relations and Graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoret. Comput. Sci. Springer (1993).

Algebraic Data Structure Refinement with the Lübeck Transformation System

Walter Dosch and Sönke Magnussen

Institute for Software Technology and Programming Languages
Medical University of Lübeck
Ratzeburger Allee 160, D–23538 Lübeck, Germany
{dosch|magnussen}@isp.mu-luebeck.de

Abstract We describe the refinement of data structures using the Lübeck Transformation System as a tool for manipulating algebraic specifications. Apart from two simple refinement steps, we present algebraic implementations as a complex refinement step based on abstraction and representation functions. We provide sufficient syntactic criteria ensuring the soundness of the transformations. We describe the user interaction within the life cycle of a specification in the transformation system.

1 Introduction

Formal methods offer a secure development process for software and hardware systems. In the transformational approach [7] a behavioural specification is systematically refined to an efficient implementation following sound transformation rules. The derived program is correct by construction without a posteriori verification. Since the formal derivation turns out to be a complex task, transformation systems [2] assist the programmer with various degrees of interaction. As standard services, they offer the safe manipulation of specifications following elementary transformation rules. Advanced transformation systems also support larger transformation steps where the user interaction is restricted to the essential design decisions. Chaining wide-spanned transformations results in compact derivations with a high degree of mechanization.

In this paper, we describe the refinement of data structures [3,8] as supported by the Lübeck Transformation System. The system analyses and transforms higher order equational specifications using various logic and algebraic rules. Apart from two simple refinement steps, we study the tool support for algebraic implementations as a complex refinement step. For the semantic refinement relation, we provide sufficient syntactic criteria which can be checked by analysis algorithms. The algebraic implementation is described using an abstraction and a representation function. Given an algorithmic description for them, the system attempts to synthesize the functions operating on the refined sort. The application conditions ensuring the soundness of the refinement step are inserted into the theory as proof obligations. The user interaction for performing the refinement steps arises from the life cycle a specification undergoes in the transformation system.

2 Theoretical Background

As specification language we use algebraic specifications with loose constructor generated semantics [13]. The theory is extended to higher order sorts and higher order terms for expressing advanced concepts of functional programming. The approach, similar to [10], additionally imposes a generation constraint for sorts to support the constructor based definition of data types. We also discriminate basic sorts and derived sorts.

The refinement of a specification enriches its signature while retaining the properties of its models. The refinement relation between specifications requires that all models of the original specification can be retrieved as subalgebras from the models of the refined specification. This notion of refinement generalizes the usual model inclusion to capture flexible transformations between data structures. Simple refinement steps are, among others, the enrichment of the signature, the addition of axioms, term rewriting, and induction.

3 The Lübeck Transformation System

The Lübeck Transformation System LTS supports the stepwise refinement of algebraic specifications [6]. The derivations head for algorithmic specifications which can be compiled into SML code. This section presents an overview of LTS surveying the life cycle of a specification and the refinement steps.

3.1 Lifecycle of a Specification

Specifications exhibit a characteristic life cycle when they are transformed. After a specification has been loaded, it resides in the system and is ready for transformation and code generation; its life cycle is illustrated in Fig. 1.

Analysis After parsing and context checking, the specification is analysed in order to inform the user about desirable or critical properties. LTS checks for each function symbol whether the defining axioms are algorithmic and complete. Orienting the equations from left to right, LTS tests confluence and termination of the resulting rewrite system. The system investigates further properties of the specification that are important for the subsequent development or its compilation into SML code. These properties assist the programmer in making future design decisions or in revising previous transformation steps.

Interaction The user starts a refinement process by selecting one of the loaded specifications as the active specification. The system enters the specification mode which allows the refinement of entire specifications. LTS also offers a fine tuning mode for transforming single axioms. The start axiom is a logical consequence of the derived axioms; the entire specification is refined by replacing the start axiom by the derived axioms. After each refinement step, the specification is analysed anew to update its properties.

The user finishes the transformation process of a specification by inserting the refined version into the collection of loaded specifications replacing the original version. Algorithmic specifications can be compiled into SML code.

3.2 Transformations

The user refines a specification by invoking sound transformations of different granularity.

In the specification mode, the transformations concern entire specifications. Elementary steps are, among others, the extension of the signature and the addition

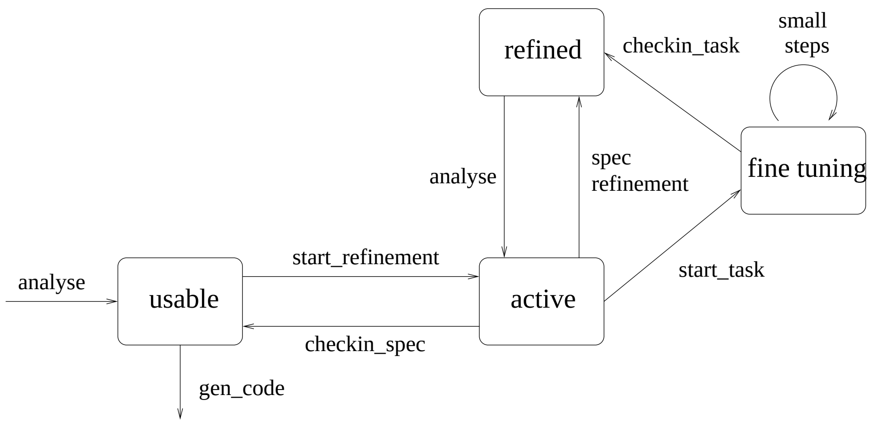


Figure 1. Life cycle of a specification

of axioms; dropping an axiom causes a proof obligation. Complex steps with wide-spanning transformations effect major changes to the entire specification. Here LTS supports the fold/unfold paradigm of recursive functions and the fusion for catamorphisms [5] .

In the fine tuning mode, the transformations refer to single axioms; elementary steps comprise rewriting, induction, generalization and simplification.

4 Refinement of Data Structures

In this section we survey the basic requirements for the refinement of data structures and study the tool support for various refinement transformations.

4.1 Basic Requirements

When refining a data structure, we realise an abstract sort by a derived sort. The refinement embeds the carrier set of the abstract sort into the carrier set of the derived sort. The transformation must not change the behaviour of the corresponding operations operating on the abstract and derived sort. Therefore we constrain the embedding by a "subalgebra condition" claiming that all models of the refined specification possess subalgebras in the models of the original specification.

The embedding can be described by a *representation function* mapping each element of the abstract carrier to an element of the derived carrier. Vice versa, an *abstraction function* maps each element of the derived carrier to an element of the abstract carrier. The commuting diagram in Fig. 2 shows the abstract sort sc , the derived sort se , the representation function $repr$, the abstraction function $abstr$, and an inner operation f_{old} on the abstract sort sc . The induced operation f on the derived sort must validate the equation $abstr \circ f = f_{old} \circ abstr$.

4.2 Enriching the Set of Constructors

The signature of a specification can be enriched by adding a function symbol to the familiy of constructors. The constructors generate the carriers of the models. So an enrichment of the constructor family will enlarge the carriers of the original

$$\begin{tikzcd} sc \arrow[r, "f_{old}"] \arrow[d, shift right, "repr"'] & sc \arrow[d, shift right, "repr"'] \\ se \arrow[r, "f"] \arrow[u, shift right, "abstr"'] & se \arrow[u, shift right, "abstr"'] \end{tikzcd}$$

Figure 2. Algebraic implementation

specification. This embedding is a sound refinement step if it validates the subalgebra condition. LTS demands that all non-constructors leading into the refined sort are completely defined by the axioms of the specification. This condition can be checked by the analysis algorithms of the system using suffcient syntactical criteria.

4.3 Reducing the Set of Constructors

The signature of a specification can be reduced by removing a function symbol from the family of constructors. In general, this will reduce the carriers of the original specification. Therefore this transformation does in general not form an embedding as imposed by the basic requirements. If the generation of the carriers does not depend on the dropped constructor symbol, the transformation is correct. Again this condition can by checked by the analysis algorithms of LTS using suffcient syntactical criteria.

4.4 Algebraic Implementations

A quite general approach to the refinement of data structures is described by the principle of algebraic implementation. In this refinement step, a constructor sort of the specification is embedded into a derived sort, and all affected operations are transformed correspondingly. In the sequel, we describe the transformations realising this complex refinement step.

Modifications and Enrichments For simplicity, we assume in this paragraph that exactly one function symbol f operates on the constructor sort sc. To avoid name clashes, the signature is changed renaming the constructor sort sc into sc_{old} and the operation f into f_{old}. Then the signature is enriched by the function symbol f operating on the derived sort, and the specification is extended by the axiom $(abstr \circ repr)(x) = x$ to gain an injective embedding. The refined operation is loosely defined by the equation $abstr \circ f = f_{old} \circ abstr$. The subalgebra property is enforced by the equation $f \circ repr = repr \circ f_{old}$.

The procedure for an algebraic implementation in LTS comprises five major user interactions.

Preparation First the user may enrich the signature or the axioms of the specification in order to obtain all relevant constituents and properties needed for the embedding.

Generating the New Specification After invoking the command for an algebraic implementation, LTS automatically renames the active specification, enlarges the signature and inserts the required axioms. These modifications establish a sound refinement step, since the application conditions are explicitly imposed as axioms. The representation and abstraction functions are just inserted into the signature postponing their axiomatic definition.

Putting the Embedding in Concrete Form The user can now insert algorithmic definitions of the representation and the abstraction functions to fix the concrete embedding.

Automatic Derivation of the Operations The system provides a strategy for automatically deriving algorithmic definitions for the new operations using the algorithmic definitions of the representation and abstraction functions. The strategy first performs a complete case analysis on the argument of the abstraction function. Then it attempts to rewrite the right-hand side into a term with the abstraction function as outermost function symbol.

Manual Completion When the strategy fails, the user can manually complete the derivation introducing additional design decisions or proving suitable propositions. Moreover, the equations arising from the subalgebra condition constitute proof obligations to be handled by the user.

5 Conclusion

In the meanwhile, the computer aided synthesis of programs from specifications has a long history. The transformation system [4] supported fold and unfold transformations using first order equations for recursively defined functions. The CIP system [1] dealt with the wide-spectrum language CIP-L supporting a large variety of transformations. A more generic approach is implemented by the TAS system [9] based on the Isabelle theorem prover [11] . The sophisticated user interface follows the principle of direct manipulation. The KIDS system [12] mechanises the synthesis of software using a knowledge base of algorithmic design principles.

The Lübeck Transformation System LTS was designed at the Institute for Software Technology and Programming Languages since 1998 and is still under development. The system is completely written in SML using the top-level environment of Moscow ML as user interface. The system combines the syntactic analysis of specifications with a transformation engine to provide wide-spanning transformation steps like data structure refinements or fusion. This leads to compact derivations showing the essential design decisions. Future work will complete the prototype implementation and proceed with a JAVA front-end as graphical user interface.

References

1. F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP: The Program Transformation System CIP-S*, volume 292 of *LNCS*. Springer, 1987.
2. F.L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformation – computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15:165–180, 1989.
3. M. Broy, B. Möller, and M. Wirsing. Algebraic implementations preserve program correctness. *Science of Computer Programming*, 7:35–53, 1986.
4. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 1(24):44–67, 1977.
5. W. Dosch and S. Magnussen. Computer aided fusion for algebraic program derivation. *Nordic Journal of Computing*, 8(3):279–297, 2001.

6. W. Dosch and S. Magnussen. Lübeck transformation system: A transformation system for equational higher-order algebraic specifications. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques: 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting, Genova, Italy, April 1-3, 2001. Selected Papers*, volume 2267 of *LNCS*. Springer, 2002. (to appear).
7. M. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Proceedings TC2 Working Conference on Program Specification and Transformation*, pages 165–195. North Holland, 1987.
8. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271 – 281, 1972.
9. C. Lüth, H. Tej, Kolyang, and B. Krieg-Brückner. TAS and IsaWin: Tools for transformational program development and theorem proving. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering FASE'99. Joint European Conferences on Theory and Practice of Software ETAPS'99*, volume 1577 of *LNCS*, pages 239– 243. Springer, 1999.
10. K. Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100:385–417, 1992.
11. L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.
12. D.R. Smith. Automating the design of algorithms. In B. Möller, editor, *Formal Program Development (IFIP TC2/WG 2.1)*, volume 755 of *LNCS*, pages 324–354. Springer, 1993.
13. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier Science Publishers, 1990.

Zufällig erzeugte BDDs: Algorithmen und Anwendungen

Ulf Milanese

Institut für Informatik und Praktische Mathematik,
Christian-Albrechts-Universität zu Kiel

Zusammenfassung Wir skizzieren, wie endliche Relationen mit Hilfe von reduzierten, geordneten binären Entscheidungsdiagrammen (ROBDDs) repräsentiert werden können. Danach präsentieren wir zwei Algorithmen zur Erzeugung zufälliger ROBDDs sowie Komplexitätsabschätzungen und Verfeinerungen für diese Algorithmen. Abschließend geben wir zwei Beispielanwendungen für die Anwendung zufällig erzeugter ROBDDs an.

1 Motivation

Reduzierte, geordnete binäre Entscheidungsdiagramme (*Reduced Ordered Binary Decision Diagrams*, kurz *ROBDDs*) finden seit Mitte der 80er Jahre verstärkt Anwendungen in der Informatik [3]. Beispielsweise werden in der Hardwareverifikation Schaltungen durch Zustandsübergangsrelationen repräsentiert, die aufgrund ihrer Größe intern nur als ROBDDs gespeichert werden können [4]. Dasselbe gilt für automatentheoretische Anwendungen, die in der Lage sind, mit sehr großen Automaten zu rechnen [8]. Für diese Anwendungen ist nur eine relativ kleine Menge von relationalen Operationen implementiert worden. So werden für die Hardwareverifikation außer der Darstellung einer Menge von Start- und Fehlerzuständen nur noch die Komposition eines Vektors mit einer Relation, der Schnitt zweier Vektoren und der Test auf Leerheit benötigt.

Das Programm RelView [1] wurde mit der Absicht entwickelt, die Operationen der Relationenalgebra für beliebige Relationen zur Verfügung zu stellen, damit relationale Programme über der Datenstruktur der Relationen ausgewertet werden können. Dabei werden im aktuellen RelView-System Relationen intern als ROBDDs gespeichert [9]. Als Beispiele für relationale Programme seien hier unter anderem Äquivalenztest, Bisimulation und Zustandsreduzierung von Transitionsrelationssystemen genannt [2].

Die Erstellung von relationalen Programmen erfordert neben der üblichen Programmverifikation (auf die in dieser Arbeit nicht eingegangen wird) auch die Durchführung von Testläufen. Dafür werden zufällige Testrelationen benötigt. Leider bestehen ROBDDs aus stark vernetzten Strukturen, die nicht zufällig zusammengesetzt werden können. Die Erzeugung einer zufälligen Relation mit einer anderen Datenstruktur und die anschließende Umwandlung in ein ROBDD haben sich aber in Tests als zu zeitaufwändig erwiesen. Aus diesem Grund war es erforderlich, Algorithmen für die zufällige Erzeugung von ROBDDs zu entwickeln.

2 Binäre Entscheidungsdiagramme

In dieser Arbeit ist nicht der Platz vorhanden, um eine umfassende Übersicht über BDDs zu geben. Es genügt für das weitere Verständnis zu wissen, dass ROBDDs zur Repräsentation von Booleschen Funktionen benutzt werden. Durch die Verwendung einer Variablenordnung und der Anwendung von Reduktionsregeln ist diese Datenstruktur eine sehr kompakte und speicherplatzsparende Darstellung für solche Funktionen. Die Booleschen Konstanten TRUE und FALSE werden in einer graphischen Darstellung von ROBDDs als Knoten mit Beschriftungen **1** und **0** dargestellt. Für weitere Details siehe [10].

Die Größe von zufälligen ROBDDs gibt C. Gröpl an [5]. In seiner Arbeit werden aber keine Verfahren zur effizienten Erzeugung angegeben, sondern nur Erwartungswerte für die Größe solcher Diagramme genannt.

3 Darstellung einer Relation als ROBDD

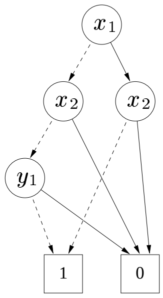
Es wird vorausgesetzt, dass der Leser mit den Grundbegriffen der Relationenalgebra vertraut ist, wie sie z.B. in [11] angegeben werden. Eine ausführliche Beschreibung, wie Relationen als ROBDDs repräsentiert werden können, findet sich in [9]. Hier soll dies an einem kleinen Beispiel illustriert werden.

Wir betrachten zwei Mengen $X = \{a, b, c, d\}$ und $Y = \{r, s\}$. Eine Relation $R : X \leftrightarrow Y, R = \{(a, r), (c, r), (c, s)\}$ wird in RELVIEW graphisch wie folgt dargestellt:

Im rechten Bild sind die Elemente des Vor- und Nachbereichs von R durch binäre Kodierungen ersetzt worden.

Die Charakteristische Funktion $\chi_R : X \times Y \to \mathbb{B}$ von R und binäre Kodierungen $c_1 : X \to \mathbb{B}^2$ und $c_2 : Y \to \mathbb{B}$ liefern eine Boolesche Funktion $f_R : \mathbb{B}^2 \times \mathbb{B} \to \mathbb{B}$, so dass gilt: $f_R(x_1, x_2, y_1) = \chi_R(c_1^{-1}(x_1, x_2), c_2^{-1}(y_1))$. Damit kann jede erfüllende Belegung der Funktion f_R durch ein Element der Relation R interpretiert werden, denn es ergibt sich $f_R(x_1, x_2, y_1) = (\overline{x_1} \wedge \overline{x_2} \wedge \overline{y_1}) \vee (x_1 \wedge \overline{x_2} \wedge \overline{y_1}) \vee (x_1 \wedge \overline{x_2} \wedge y_1)$.

Wenn als Variablenordnung $x_1 < x_2 < y_1$ gewählt wird, erhalten wir das folgende ROBDD:



Die Repräsentation von Relationen auf diese Weise bedingt, dass ein ROBDD nicht eindeutig eine einzige Relation darstellt. Außerdem liefert diese Konstruktion im allgemeinen nur partielle Boolesche Funktionen, falls Vor- oder Nachbereich einer Relation nicht exakt als Mächtigkeiten Zweierpotenzen besitzen. Diese Probleme konnten wir lösen, indem die Dimensionen der repräsentierten Relation explizit gespeichert werden und Undefiniertheiten der Funktionen durch Nichterfüllung ersetzt werden. Diese Vorgehensweisen werden in [9] ausführlich behandelt.

4 Algorithmen zur Erzeugung

Im folgenden gehen wir davon aus, dass zur Kodierung des Vor- und Nachbereichs einer zu erzeugenden Relation n Variablen benötigt werden. Die Wahrscheinlichkeit, dass ein Eintrag in der Relation vorhanden ist, soll $p \in [0 \ldots 1]$ betragen.

Für die Erzeugung eines zufälligen ROBDDs bieten sich zwei Vorgehensweisen an:

1. Es werden $p * 2^n$ Belegungen zufällig gewürfelt und durch $\vee$ verknüpft. Falls $p \leq 0,5$ ist, werden die erfüllenden Belegungen erzeugt. Ansonsten werden die nicht-erfüllenden Belegungen mit der Wahrscheinlichkeit $1 - p$ generiert und mittels Negation und $\wedge$ verknüpft.
2. Für jede mögliche Belegung wird mit Wahrscheinlichkeit p entschieden, ob die Belegung zu TRUE ausgewertet wird.

Bei der ersten Vorgehensweise können wir annehmen, dass nur neue Belegungen gewürfelt werden. Falls nämlich zu der gegebenen Variablenordnung eine Variable x_i existiert, deren Belegung zum ROBDD-Knoten **1** führt, wird diese Variable in der Belegung negiert. Außerdem benötige die Erzeugung einer Belegung $\mathcal{O}(n)$ Rechenschritte.

Das ROBDD, das eine Belegung repräsentiert, enthält genau $n + 1$ Knoten (n Variablen und eine Konstante). Für die Operationen $\vee$ (bzw. $\wedge$) wird von der Wurzel bis zur Konstanten genau ein Pfad im Zwischenresultat verfolgt, von der Konstanten bis zur Wurzel werden höchstens n neue Knoten erzeugt und eingebunden. Also ist der Rechenaufwand für das Einfügen einer Belegung $\mathcal{O}(n)$. Damit ergibt sich als Gesamtaufwand für die Erzeugung der zufälligen Funktion $\mathcal{O}(n * p' * 2^n)$, wobei p' das Minimum von p und $1 - p$ sei.

Die Idee bei der zweiten Methode zur Erzeugung eines zufälligen ROBDDs ist die Generierung von 2^n konstanten ROBDD-Knoten und deren rekursive Verknüpfung.

Falls im rekursiven Aufruf zwei ROBDDs identisch sind, kommt eine Reduktionsregel zur Anwendung, ansonsten wird ein neuer Knoten mit nächsthöherem Index erzeugt. Die Generierung eines konstanten Knotens benötigt eine Rechenzeit, die in $\mathcal{O}(1)$ liegt. Die Verknüpfung mit Hilfe eines IfThenElse-Operators geschieht ebenfalls in konstanter Zeit. Damit liegt der Gesamtaufwand zur Erzeugung des zufälligen ROBDDS in $\mathcal{O}(2^n)$.

Es ist ziemlich einfach zu entscheiden, welches Verfahren im Einzelfall anzuwenden ist. Bezeichne p' das Minimum von p und $1-p$. Dann gilt, dass das erste Verfahren dem zweiten vorzuziehen ist, falls $p' * n \leq 1$ ist, d.h. $p' \leq \frac{1}{n}$.

Bezeichne $B^i_{n,p}$ die Menge der erfüllenden Belegungen des Resultat-ROBDDs, welches durch die i-te Methode erzeugt wird. Dann gilt $|B^1_{n,p}| = 2^n * p$ und $E(|B^2_{n,p}|) = 2^n * p$. Dies bedeutet, dass wir bei der ersten Vorgehensweise eine exakte Aussage über die Anzahl der erfüllenden Belegungen treffen können, während wir bei der zweiten Methode nur einen Erwartungswert angeben können.

Für den Fall, dass die Größe des Vor- oder Nachbereichs der zu erzeugenden Relation keine Zweierpotenz ist, müssen diese beiden Algorithmen verfeinert werden. Das erste Verfahren lässt sich leicht auf den Definitionsbereich der Relation einschränken.

Bei der zweiten Methode wird aber ein ROBDD erzeugt, das mit hoher Wahrscheinlichkeit erfüllende Belegungen enthält, die undefinierte Einträge in der Relation repräsentieren. Um dies zu verhindern, muss das zweite Verfahren dahingehend verändert werden, dass es ohne Rekursion auskommt. Dies wurde von uns realisiert, indem die Relationsgröße als Liste von Einsen und Nullen als zusätzliche Parameter angegeben wird. Falls die Bereichsgrößen Zweierpotenzen sind, ändert sich nichts am Verfahren. Ansonsten werden beim Erreichen des Grenzwertes in dem Zwischenergebnis alle noch nicht vervollständigten ROBDD-Knoten mit der Konstanten **0** vereinigt.

Die Kodierung der Relationsgröße konnte mit einer Laufzeit in $\mathcal{O}(n)$ implementiert werden. Die Überprüfung, ob die Mächtigkeit des Vor- oder Nachbereich eine Zweierpotenz ist, kann in $\mathcal{O}(n)$ erfolgen. Der Test auf Erreichung dieser Werte konnte bitweise realisiert werden und liegt insgesamt in $\mathcal{O}(n)$. Damit ergibt sich trotz der Verfeinerung weiterhin ein Gesamtaufwand, der unverändert in $\mathcal{O}(2^n)$ liegt.

5 Anwendungen

Ein Anwendungsgebiet für Relationen, die durch zufällige ROBDDs repräsentiert werden, ist das Testen von Invarianten. Dabei liege ein (z.B. relationales) Programm vor. Für den nötigen Korrektheitsbeweis wird eine Invariante vermutet, deren Gültigkeit nicht offensichtlich ist. Bevor nun Aufwand in den Beweis dieser Invariante gesteckt wird, kann diese zuerst an einer gewissen Zahl von Beispielrelationen getestet werden. Falls ein Test mißlingt, muß keine Arbeit in den Beweis gesteckt werden.

Ein Beispiel für diese Arbeitsweise findet sich in [2]. Die gefundene (und bewiesene) Invariante hat dort den Korrektheitsbeweis erheblich verkürzt.

Eine weitere Anwendung ist die automatische Generierung von Testgraphen. Graphen können als homogene (quadratische) Relationen dargestellt werden. Seien $G = (V, E)$ und $R = (V, E')$ zwei Graphen mit $E' \subseteq E$. R heißt **transitive Reduktion** von G, falls gilt $R^* = G^*$ und $\forall E'' \subset E' : (V, E'')^* \neq R^*$. Da die Berechnung einer Lösung für dieses Problem $\mathcal{NP}$-vollständig ist, werden für viele Anwendungsbereiche Annäherungen berechnet. Wenn zwei Approximations-Algorithmen zur Berechnung einer transitiven Reduktion vorliegen, stellt sich die Frage, welcher Algorithmus im allgemeinen schneller ist oder kleinere Lösungen als Ergebnis liefert. Diese Algorithmen können an einer repräsentativen Anzahl von Beispielgraphen getestet werden. Dies ermöglicht den Vergleich der Anzahl der Kanten in den Lösungen und der benötigten Rechenzeiten. So konnte z.B. Kasper in [6] nachweisen, dass der Algorithmus zur Berechnung einer transitiven Reduktion von Simon [7] aus dem Jahre 1990 fehlerhaft ist.

Literatur

1. R. Behnke, R. Berghammer, and P. Schneider. Machine support of relational computations: The Kiel RelView System. Technical Report 9711, Institut für Informatik und Praktische Mathematik, Universität zu Kiel, Kiel, 1997.
2. R. Berghammer, T. Hoffmann, B. Leoniuk, and U. Milanese. Prototyping and programming with relations. Electronic Notes in Theoretical Computer Science.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

4. J. R. Burch, E. M. Clarke, D. L. Dill, and K. L. McMillan. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference, Orlando, FL, 24-28 June 1990*, pages 46–51, 1990.
5. C. Gröpl. *Binary Decision Diagrams for Random Boolean Functions.* PhD thesis, Humboldt-Univ. zu Berlin, 1999.
6. C. Kasper. Untersuchung von Algorithmen für transitive Reduktionen und minimale Äquivalenzgraphen. Master's thesis, Christian-Albrechts-Universität zu Kiel, Institut für Inforamatik und Praktische Mathematik, 2001.
7. Samir Khuller, Balaji Raghavachari, and Neal Young. Approximating the minimum equivalent digraph. *SIAM Journal on Computing*, 24(4):859–872, August 1995.
8. Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.
9. B. Leoniuk. *ROBDD-basierte Implementierung von Relationen und relationalen Operationen mit Anwendungen.* PhD thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 2001.
10. C. Meinel and T. Theobald. *Algorithmen und Datenstrukturen im VLSI-Design.* Springer, 1989.
11. G. Schmidt and T. Ströhlein. *Relations and Graphs.* ETACS Monographs on Theoretical Computer Science. Springer-Verlag, 1993.

A Mechanically Verified Bootstrap Compiler*

Axel Dold and Vincent Vialard

Fakultät für Informatik
Universität Ulm
D-89069 Ulm, Germany
Fax: +49/(0)731/50-24119
{dold|vialard}@informatik.uni-ulm.de

1 Introduction

The use of computer based systems for safety-critical applications requires high dependability of the software components. In particular, it justifies and demands the verification of programs typically written in high-level programming languages. Correct program execution, however, crucially depends on the correctness of the binary machine code executable, and therefore, on the correctness of system software, especially compilers. As already noted in 1986 by Chirica and Martin [2], full compiler correctness comprises both the correctness of the compiling specification (with respect to the semantics of the languages involved) as well as the correct implementation of the specification.

Verifix [6,9] is a joint German research effort of groups at the universities Karlsruhe, Kiel, and Ulm. The project aims at developing innovative methods for constructing provably correct compilers which generate efficient code for realistic, practically relevant programming languages. These realistic compilers are to be constructed using approved development techniques. In particular, even standard unverified compiler generation tools (such as Lex or Yacc) may be used, the correctness of the generated code being verified at compile time using verified program checkers [7]. *Verifix* assumes hardware to behave correctly as described in the instruction manuals.

In order not to have to write the verified parts of the compiler and checkers directly in machine code, a fully verified and correctly implemented *initial compiler* is required, for which efficiency of the produced code is not a priority. The initial correct compiler to be constructed in this project transforms ComLisp programs into binary Transputer code. ComLisp is an imperative proper subset of ANSI-Common Lisp and serves both as a source and implementation language for the compiler. The construction process of the initial compiler consists of the following steps:

- define syntax and semantics of appropriate intermediate languages.
- define the compiling specification, a relation between source and target language programs and prove (with respect to the language semantics) its correctness according to a suitable correctness criterion.
- construct a correct compiler implementation in the source language itself (a transformational constructive approach is applied which builds a correct implementation from the specification by stepwise applying correctness-preserving development steps [5]).

* This research has been funded by the Deutsche Forschungsgemeinschaft (DFG) under project "*Verifix*".

- use an existing (unverified) implementation of the source language (here: some arbitrary Common Lisp compiler) to execute the program. Apply the program to itself and bootstrap a compiler executable. Check syntactically, that the executable code has been generated according to the compiling specification. For this last step, a realistic technique for low level compiler verification has been developed which is based on rigorous a posteriori syntactic code inspection [8,11]. This closes the gap between high-level implementation and executable code.

The size and complexity of the verification task in constructing a correct compiler is immense. In order to manage it, suitable mechanized support for both specification and verification is necessary. We have chosen the PVS specification and verification system [16] to support the verification of the compiling specification and the construction process of a compiler implementation in the source language.

In this extended abstract we briefly sketch the mechanical verification of the compiling specification of the first compilation phase from ComLisp to the stack-intermediate language SIL, the first of a series of intermediate languages used to compile ComLisp programs into binary Transputer machine code:

$$\text{ComLisp} \rightarrow \text{SIL} \rightarrow \text{C}^{\text{int}} \rightarrow \text{TASM} \rightarrow \text{TC}$$

For a detailed description of the formalization and verification of the first compilation phase consult [4].

2 ComLisp and SIL

A ComLisp program consists of a list of global variables, a list of possibly mutual recursive function definitions, and a main form. ComLisp forms (expressions) include the *abort* form, s-expression constants, variables, assignments, sequential composition (*progn*), conditional, while loop, call of user defined functions, call of built-in unary (*uop*) and binary (*bop*) ComLisp operators, local let-blocks, *list∗* operator (constructing a s-expression list from its evaluated arguments), case-instruction, and instructions for reading from the input sequence and writing to the output. The ComLisp operators include the standard operators for lists (e.g. *length*), type predicates for the different kinds of s-expressions, and the standard arithmetic operations (e.g. $+, *, \mathit{floor}$). The only available datatype is the type of s-expressions which are binary trees built with constructor “cons”, where the leaves are either integers, characters, strings, or symbols. The abstract syntax of ComLisp is given as follows:

$$\begin{array}{lll}
p & ::= & x_1, \ldots, x_k; f_1, \ldots, f_n; e \\
f & ::= & h(x_1, \ldots, x_m) \leftarrow e \\
e & ::= & \mathit{abort} \mid c \mid x \mid x := e \mid \mathit{progn}(e_1, \ldots, e_n) \mid \mathit{if}(e_1, e_2, e_3) \mid \mathit{while}(e_1, e_2) \mid \\
 & & \mathit{call}(h, e_1, \ldots, e_n) \mid \mathit{uop}(e) \mid \mathit{bop}(e_1, e_2) \mid \mathit{let}(x_1 = e_1, \ldots, x_n = e_n; e) \mid \\
 & & \mathit{list}{*}(e_1, \ldots, e_n) \mid \mathit{cond}(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n) \mid \\
 & & \mathit{read_char} \mid \mathit{peek_char} \mid \mathit{print_char}(e)
\end{array}$$

The static semantics of ComLisp programs (dealing, among other things, with the declaration of variables), function definitions, and forms is specified by means of several well-formedness predicates.

For the intermediate languages occurring in the different compilation phases of the ComLisp to Transputer compiler, a uniform relational semantics description has been chosen. The (dynamic) semantics of ComLisp is defined in a structural operational way by a set of inductive rules for the different ComLisp forms (*big-step semantics* or *evaluation semantics*). A ComLisp *state* is a triple consisting of an (infinite) input sequence (stream) of characters, an output list of characters, and the variable state which is a mapping from identifiers to values (s-expressions). ComLisp forms are expressions with side-effects, that is, they denote state transformers transforming states to pairs of result value and result state. The semantics of a ComLisp program is given by the input/output behavior of the program defined by a relation $P_{\mathrm{sem}_{\mathrm{CL}}}(p)(is, ol)$ between input streams is and output lists ol.

SIL, the stack intermediate language, is a language with parameterless procedures and s-expressions as available datatype. Programs operate on a runtime stack with frame-pointer relative addresses. A SIL program consists of a list of parameterless procedure declarations and a main statement. There are no variables, only memory locations and the machine has statements for copying values from the global to the local memory and vice versa. For example, $copy(i, j)$ copies the content at stack relative position i to relative position j.

$$
\begin{array}{lll}
p & ::= & f_1, \ldots, f_n; s \\
f & ::= & h \leftarrow s \\
s & ::= & abort \mid copyc(c, i) \mid copy(i, j) \mid gcopy(g, i) \mid copyg(g, i) \mid \\
 & & itef(i, s_1, s_2) \mid sq(s_1, \ldots, s_n) \mid fcall(h, i) \mid uop(i) \mid bop(i) \mid \\
 & & while(i, s_1, s_2) \mid read\_char(i) \mid peek\_char(i) \mid print\_char(i) \mid list{*}(n, i)
\end{array}
$$

SIL statements denote state transformers, where a SIL state consists of the input stream, the output list, the global memory (a list of s-expressions), and the local memory (consisting of the frame pointer $base : Nat$ and the stack, a function from natural numbers to s-expressions). As for ComLisp, the semantics of a SIL program is its I/O behavior (predicate $P_{\mathrm{sem}_{\mathrm{SIL}}}(p)(is, ol)$).

3 Compiling ComLisp to SIL

The compilation from ComLisp to SIL generates code according to the stack principle and translates parameter passing to statements which access the data stack. For a given expression e, a sequence of SIL instructions is generated that computes its value and stores it at the top of the stack (relative position k in the current frame). The parameters $x_1, \ldots, x_n$ of a function are stored at the bottom of the current frame (at relative positions $0, \ldots, n-1$). A SIL function call $fcall(h, i)$ increases the frame pointer $base$ by i which is reset to its old value after the call and local variables introduced by *let* are represented within the current frame. For each syntactical ComLisp category, a compiling function is specified ($\mathcal{C}_{\mathrm{prog}}(p)$ denotes the compilation function for programs).

4 Correctness of the Compilation Process

The notion of correctness used in *Verifix* is the preservation of the observable behavior up to resource limitations. In our case correctness of the compilation process is stated as follows: for any well-formed ComLisp program p, whenever the semantics of the compiled program is defined for some input stream is and output list ol, this is also the case for p for the same is and ol:

Theorem 1 (Correctness of Program Compilation).
$\forall p, is, ol.\ wf_{\mathrm{program}}(p) \Rightarrow (P_{\mathrm{sem}_{\mathrm{SIL}}}(\mathcal{C}_{\mathrm{prog}}(p))(is)(ol) \Rightarrow P_{\mathrm{sem}_{\mathrm{CL}}}(p)(is)(ol))$

Unfolding $P_{\mathrm{sem}_{\mathrm{SIL}}}$ and $P_{\mathrm{sem}_{\mathrm{CL}}}$, the semantics of forms and corresponding SIL statements have to be compared. In particular, this requires relating source and target language states. ComLisp forms denote state transformers transforming a state into a result value and a result state (if defined) $\sigma \rightarrow_e (v, \sigma')$. On the other hand, SIL statements denote ordinary state transformers $s \rightarrow_s s'$. Two relations are required: one relation ρ_{in} relates ComLisp input states σ with SIL states s, while the other relation ρ_{out} relates ComLisp output states (v, σ') with SIL states s'. The main obligation therefore is to prove the correctness property for forms (illustrated in Figure 1) which includes additional state invariants for the source and target level (omitted here).

$$\begin{tikzcd}[column sep=huge, row sep=large]
state_{\mathrm{CL}} \ni \sigma \arrow[r, "\rightarrow_e"] \arrow[d, dash, "\rho_{\mathrm{in}}"'] \arrow[dr, phantom, "\circlearrowright"] & (v, \sigma') \in SExpr \times state_{\mathrm{CL}} \arrow[d, dash, "\rho_{\mathrm{out}}"] \\
state_{\mathrm{SIL}} \ni s \arrow[r, "\rightarrow_s"] & s' \in state_{\mathrm{SIL}}
\end{tikzcd}$$

Figure 1. Correctness property for the compilation of ComLisp forms

5 PVS Formalization and Verification

The specification language of the PVS verification system is based on classical higher-order logic with a rich type system including dependent types. In addition, the PVS system provides support tools and an interactive sequent-calculus based proof checker that has a reasonable amount of theorem proving capabilities. A strategy language enables to combine atomic inference steps into more powerful proof strategies allowing to define reusable proof methods.

Abstract syntax, static and dynamic semantics of the languages, the compiling functions and the compilation theorems have to be formalized. For abstract syntax, the PVS abstract data type (ADT) construct is used. For the dynamic semantics, a set of rules is represented as an inductive PVS relation which combines all the rules in one single definition. The main obligation to prove is the correctness property for forms which is proved by well-founded induction using a specific termination measure. To suitably manage this proof, for each kind of form a separate compilation theorem is introduced. Although strategies for parts of the proofs have been developed, the number of manual steps is quite high and shows that this verification task is by no means trivial. All the proofs have been completely accomplished using PVS.

It is hard to give an estimation of the amount of work invested in the final verification, since we started the verification on a smaller subset of ComLisp in order to experiment with different styles of semantics and find the necessary invariants, and then incrementally extended this subset and tried to rerun and adapt the already accomplished proofs. A coarse estimation of the total formalization and verification effort required for the compiling specification for all 4 compilation phases is about 3 person-years.

6 Related Work

Verification of compiler correctness is a much-studied area starting with the work by McCarthy and Painter in 1967 [13], where a simple compiler for arithmetic expressions has been proved correct. Many different approaches have been taken since then, usually with mechanized support to manage the complexity of the specifications and the proofs, for example [1, 3, 12, 14, 17]. Most of the approaches only deal with the correctness of the compiling specification, while the approach taken in the *Verifix* project also takes care of the implementation verification, even on the level of binary machine code. Another difference of our approach is that we are concerned with the compilation of "realistic" source languages and target architectures. A ComLisp implementation of the ComLisp compiler as well as a binary Transputer executable is available.

Notable work in this area with mechanized support is CLInc's verified stack of system components ranging from a hardware-processor up to an imperative language [14]. Both the compiling verification and the high-level implementation (in ACL2 logic which is a LISP subset) have been carried out with mechanized support using the ACL2 prover. Using our compiler, correct binary Transputer code could be generated.

The impressive VLISP project [10] has focused on a correct translation for Scheme. However, although the necessity of also verifying the compiler implementation has been expressed this has explicitly been left out. Proofs were accomplished without mechanized support.

P. Curzon [3] considers the verification of the compilation of a structured assembly language, Vista, into code for the VIPER microprocessor using the HOL system. Vista is a low-level language including arithmetic operators which correspond directly to those available on the target architecture.

The compilation of PROLOG into WAM has been realized through a series of refinement steps and has been mechanically verified using the KIV system [18]. A (small-step) ASM semantics is used for the languages.

7 Concluding Remarks

The formalization and formal verification of the compiling specification for the bootstrap compiler is an ongoing effort. Besides the verification of the first compilation phase, the verification of the second phase, the translation from SIL to C^{int}, where s-expressions and their operators are implemented in linear integer memory (classical data and operation refinement), is also completed. Current work is concerned with the verification of the compiler back-end, namely, the compilation from C^{int} into abstract Transputer assembler code TASM. The standard control structures of C^{int} must be implemented by conditional and unconditional jumps, and the state space must be realized on the concrete Transputer memory. The verification of the last compilation phase, where abstract Transputer assembler is compiled into binary

Transputer code (TC) has already been accomplished following approved verification techniques [15]: starting from a (low-level) base model of the Transputer, where programs are a part of the memory, a series of abstraction levels is constructed allowing different views on the Transputer's behavior and the separate treatment of particular aspects.

We have demonstrated that the formal, mechanized verification of a non-trivial compiler for a (nearly) realistic programming language into a real target architecture is feasible with state-of-the-art prover technology.

Acknowledgements

The construction of the bootstrap compiler is joint work with the project partners from the university of Kiel. We thank them for their constructive collaboration and many discussions on this subject.

References

1. E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd Int. Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*. Springer, 1998.
2. L.M. Chirica and D.F. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.
3. Paul Curzon. The Verified Compilation of Vista Programs. Internal Report, Computer Laboratory, University of Cambridge, January 1994.
4. A. Dold and V. Vialard. A Mechanically Verified Compiling Specification for a Lisp Compiler. In R. Hariharan, M. Mukund, V. Vinay editors, *Proc. of the 21st Annual FSTTCS Conference*, Bangalore, India, December 2001, volume 2245 of *LNCS*. (an extended version is available from www.informatik.uni-ulm.de/ki/fsttcs01.html)
5. Axel Dold. *Formal Software Development using Generic Development Steps.* Logos-Verlag, Berlin, 2000. Dissertation, Universität Ulm.
6. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The Verifix Approach. In *Proceedings of the Poster Session of CC'96 - International Conference on Compiler Construction.* ida, 1996. TR-Nr.: R-96-12.
7. Wolfgang Goerigk, Thilo Gaul, and Wolf Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, Malente, 1998. Springer Verlag.
8. Wolfgang Goerigk and H. Langmaack. Compiler Implementation Verification and Trojan Horses. In D. Bainov, editor, *Proc. of the 9th Int. Colloquium on Numerical Analysis and Computer Sciences with Applications, Plovdiv, Bulgaria*, 2000.
9. G. Goos and W. Zimmermann. Verification of Compilers. In B.Steffen E.-R. Olderog, editor, *Correct System Design*, volume 1710 of *LNCS*, pages 201–230. Springer-Verlag, 1999.
10. J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A Guide to VLISP, A Verified Programming Language Implementation. Technical Report M92B091, The MITRE Corporation, Bedford, MA, September 1992.
11. Ulrich Hoffmann. *Compiler Implementation Verification through Rigorous Syntactical Code Inspection.* PhD thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Kiel, 1998.

12. J.J. Joyce. A Verified Compiler for a Verified Microprocessor. Technical Report 167, University of Cambridge, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England, March 1989.
13. J. McCarthy and J.A. Painter. Correctness of a Compiler for Arithmetical Expressions. In J.T. Schwartz, editor, *Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.
14. J S. Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5(4), 1989.
15. Markus Müller-Olm. *Modular Compiler Verification*, volume 1283 of *LNCS*. Springer Verlag, Berlin, Heidelberg, New York, 1997. PhD Thesis.
16. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
17. W. Polak. *Compiler Specification and Verification*, volume 124 of *LNCS*. Springer-Verlag, 1981.
18. Gerhard Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Unversität Ulm, 1999.

Syntactic Type Soundness Results for the Region Calculus*

Cristiano Calcagno, Simon Helsen, and Peter Thiemann

Queen Mary, University of London and University of Freiburg

1 Introduction and Motivation

Memory management for dynamic data structures is a problem in programming. While memory allocation is dictated by the problem at hand, there is considerable freedom in memory deallocation. If deallocation happens too late, the program suffers from memory bloat and space leaks, which impede performance. If deallocation happens too early, there might be dangling pointers into deallocated memory. Dereferencing a dangling pointer is unsafe and can lead to a crash, or worse, to wrong results.

Some languages (like C or Pascal) leave the deallocation problem entirely to the programmer, whereas others (like Lisp, Smalltalk, Java, and ML) perform automatic deallocation by incorporating a trace-based garbage collector into the runtime system. While the programmer-based solution is immensely error-prone, programs can in principle be tuned for optimal memory use. Traditional garbage collection avoids a large class of errors, but it has some problems, too. Since the garbage collector is, in general, unaware of the semantics of the running program, it must preserve all pointers reachable from a given set of root pointers. This set is a conservative approximation of the set of pointers that will actually be used by the program. As a consequence, deallocation might happen too late, which can lead to space leaks. In addition, trace-based garbage collection takes extra, non-productive time and can cause erratic pauses in the execution of programs, hampering its use for real-time applications. Finally, inter-operability between garbage collected languages and non-garbage collected languages is difficult.

The region calculus of Tofte and Talpin [13, 14] (which we refer to as TTRC) provides an alternative method of memory management for the functional language ML [10]. It is used as an intermediate language in an ML compiler, the ML-kit [2,3, 12–14]. The basic idea of the region calculus is to split memory into regions that are allocated in a stack-like manner, directed by a construct of the language. Deallocation is instantaneous, it just pops the topmost region from the stack. Using this method, it is possible to implement ML without a trace-based garbage collector. In some instances, the region calculus can prove that a pointer is semantically dead, even though it is still reachable by the program. In these cases, the region it points to can be safely deallocated, something trace-based garbage collectors cannot do.

2 Related Work

The first proof of consistency, or type soundness, for the region calculus as it is given by Tofte and Talpin [14] is quite complicated and uses rule-based co-induction.

* Extended abstract: details are to be published in *Information and Computation* [5].

The source of complication is twofold. First, Tofte and Talpin prove two properties at the same time: type soundness and *translation soundness*. The latter property guarantees that there is some relation between a non-region annotated value and its region-based counterpart. In this paper, we focus on the problem of type soundness, ie. the property which guarantees that regions are not deallocated while they are still in use.

The second source of complication is due to the co-inductive definition of their consistency relation, required because of the loss of information when deleting a region from the store in their big-step semantics. Their safety relation not only requires a co-inductive proof, but is rather complex and lacks intuition of why deallocation safety is obtained.

Recently, alternative type-soundness proofs for the region calculus have been proposed.

1. Crary, Walker, and Morrisett [6] provide an indirect soundness proof by translating the region calculus into their capability calculus. The capability calculus has a sophisticated type-and-effect system that supports safe allocation and deallocation of regions in an arbitrary order. This added flexibility may lead to a better use of memory at runtime, since there are cases where a region may be deallocated earlier than in the region calculus. They provide a syntactic soundness proof for the capability calculus.
2. Banerjee, Heintze, and Riecke [1] translate the region calculus into $F_{\#}$, an extension of the polymorphic lambda calculus with a special type constructor for encapsulation. They construct an original denotational model for their calculus and give a semantic soundness proof based on the model.
3. Dal Zilio and Gordon [16] modify the operational semantics of Tofte and Talpin so that it also keeps track of deallocated regions. This extra information allows an inductive definition of the consistency relation and an inductive correctness proof. Then they go on to show that this result is a consequence of a more general result for a typed π-calculus with name groups. This is shown by using a translation from the region calculus into the typed π-calculus with name groups.
4. Helsen and Thiemann [9] define a store-less small-step operational semantics for the region calculus and prove type soundness using the syntactic method of Wright, Felleisen, and Harper.
5. Calcagno [4] defines a high-level big-step operational semantics and proves type soundness for it. Calcagno formally relates the high-level semantics to the original low-level semantics of TTRC.

3 Contribution and Overview

The present research [5] is based on the work of Calcagno, Helsen, and Thiemann [4, 9]. We give a simplified account of a *store-less region calculus* (abbreviated SRC), using the reduction-style formulation pioneered by Plotkin [11]. Its syntactic type soundness is formulated without proofs and without the treatment of polymorphism and recursion, which can be found elsewhere [9].

While the store-less formulation is extremely simple and elegant, it is desirable to model a calculus with references and destructive update. Therefore, we introduce a new calculus with an explicitely passed store: the *imperative region calculus* or IRC. This calculus extends SRC (and TTRC) with operations on references, as they are actually implemented in the ML-kit [3]. We also give a small-step operational semantics, similar in spirit to the definition of the store-less region calculus.

Then, using the syntactic approach of Wright and Felleisen [15], in a variation pioneered by Harper [7], we prove type soundness of IRC without the standard treatment of polymorphism and recursion. Adding polymorphism and recursion makes the proofs more technical, but it does not require new insights. The resulting proofs all follow a relatively simple inductive pattern, and are therefore considerably easier than the co-inductive proofs of Tofte and Talpin.

In previous work, Calcagno [4] proves type soundness of TTRC by defining a store-less big-step operational semantics, which is parametric in a set of currently allocated regions. He proves his store-less semantics equivalent to TTRC.

Inspired by this work, we show the equivalence of TTRC with IRC, as well as the equivalence of IRC and SRC. However, instead of relating two big-step semantics, we relate a big-step semantics (TTRC) with a small-step semantics (IRC) on the one hand and two small-step semantics (IRC and SRC) on the other hand. The former result leads to type-soundness of TTRC. In these equivalences, we ignore the reference operations of IRC for simplicity of the presentation.

References

1. Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proc. of the 14th Annual IEEE symposium on Logic in Computer Science*, pages 88–97, Trento, Italy, July 1999. IEEE Computer Society Press.
2. Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 58:299–392, 2001.
3. Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, Fla., January 1996. ACM Press.
4. Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In Hanne Riis Nielson, editor, *Proc. 28th Annual ACM Symposium on Principles of Programming Languages*, pages 155–165, London, England, January 2001. ACM Press.
5. Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, to appear.
6. Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In Alex Aiken, editor, *Proc. 26th Annual ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, USA, January 1999. ACM Press.
7. Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also note [8].
8. Robert Harper. A note on: "A simplified account of polymorphic references". *Information Processing Letters*, 57(1):15–16, January 1996. See also [7].
9. Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In Alan Jeffrey, editor, *ACM Workshop on Higher Order Operational Techniques in Semantics*, volume 41(3) of *Electronic Notes in Theoretical Computer Science*, pages 1–20, Montreal, Canada, September 2000. Elsevier Science.
10. Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
11. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

12. Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.
13. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, OG, January 1994. ACM Press.
14. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
15. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
16. S. Dal Zilio and Andrew Gordon. Region analysis and a pi-calculus with groups. In *Proceedings of MFCS '00*, volume 1893 of *Lecture Notes in Computer Science*, pages 1–20, 2000.

Program Transformation with Term-Graph Patterns

Frank Derichsweiler

University of the German Federal Armed Forces
Department of Computer Science, Institute for Software Technology
Werner-Heisenberg-Weg 39, D-85577 Neubiberg
deri@informatik.unibw-muenchen.de

Abstract In the context of the integrated abstract transformational program development environment HOPS we introduce the notion of term-graph patterns for easily specifying powerful transformation strategies, which automatise program transformation as a sequence of transformation rule applications.

1 Introduction

In many contexts (e. g. [1–3, 5, 10, 12–14]) program transformation is a feasible and fruitful approch to developing and / or improving programs.
In this paper, we discuss necessities for mechanising program transformation by applying powerful transformation strategies within the integrated development environment HOPS, in which programs are represented as term graphs.
After describing the basic ideas of HOPS we focus on the requirements for easily specifying expressive transformation strategies. Term-graph patterns will be introduced as the key-component for specifying and controlling mechanised application of transformation strategies.

2 HOPS

The **H**igher **O**bject **P**rogramming **S**ystem HOPS is a graphically interactive term graph programming system designed for transformational program development, see also [2, 8, 9, 19]. (A prototypical implementation in the programming language Smalltalk is available.)

In the spirit of Literate Programming [11], HOPS modules are documents containing program fragments. In HOPS, these are mostly declarations, attribution definitions, further on transformation rules and strategies — declarations and rules are created and manipulated as *term graphs*.

HOPS manipulates arbitrary second-order term graphs, where all the structure usually encoded via name and scope is made explicit. Term graphs in HOPS therefore feature nameless variables, explicit variable binding (to denote which node binds which variable), explicit variable identity (to denote which nodes stand for the same variable) and metavariables with arbitrary arity; for a detailed introduction to this term graph concept see [7].

Every HOPS term DAG is partitioned into an object and a type layer. The users build their programs within the object layer using bricks from the module system. The type layer is automatically calculated on the fly, i. e., during editing. This ensures that the user can always inspect the actual typing of a used brick. It is not possible to construct an untypeable term DAG within HOPS. The type system is deterministic and similar to the polymorphic type system of ML (c. f. [7]).

All nodes in the term DAG are marked with brick labels, which identify the used constructors. Further on, the outgoing edges, the cardinality of which corresponds to the arity of the constructor used, have edge labels to identify the ordering of the parameters of the constructor.

Only bricks for variables are predefined in the HOPS module system. On top of these, and within the constraints of the typing system, a wide variety of languages may be defined by the user. Example languages are provided. The most elaborate one follows the functional programming paradigm and is close to Haskell [4] in its spirit.

Having no hard-coded language is considered to be one of the advantages of HOPS (only the variables are fixed within the implementation): Different languages for different domains and / or levels of abstraction give flexibility. The HOPS user declares and uses the bricks which are appropriate for his situation. Therefore HOPS is intended to be a user-friendly framework for specifying and using different domain specific languages. For example switching between different levels of abstraction can be performed by applying transformation operations.

3 Program Transformation

Program transformation in HOPS is an interactive process and not considered as a "black box" operation, as in in other systems, for instance Stratego/System S [16–18]. Church-Rosser-properties of the transformation system are not within our main focus ([15] discusses the minor importance of these theoretical concepts for practical applications of program transformation).

In the context of HOPS we use transformation rule application and maximal-identification as primitive transformation operations. The latter means searching for common sub-expressions and identifying them within the term graph.
A HOPS transformation rule is a term graph with two explicitly marked nodes, the left and right rule-side node. In order to apply a transformation rule, we need a matching homomorphism, which maps the sub-term-graph induced by the left rule node to the the term graph in question, mapping the left rule node to the node at which the rule is applied. The result of the application is computed by constructing the image of the right rule side (i. e., the sub-term-graph induced by the right rule node) and then replacing the image of the left rule side by this term graph. Details about the rule mechanism and its theoretical foundation are given in [6].

Mechanised program transformation within HOPS means to specify and apply a sequence of transformation rule applications and maximal identification steps in different areas of the term graph under examination. This is done by defining transformation strategies. Another view of a transformation strategy is that of a function which schedules the application of (different) transformation rules at (different) nodes of the term graph in question interspersed by maximal identification operations.
The transformation strategy is constructed from the two compontents *navigation* and *action*. The navigation part controls at which node transformation operation(s) will be performed; the action part determines those transformation operation(s). The interleaving of the two components during strategy application controls the scheduling.

4 Term-Graph Pattern

A naive approach for the navigation component uses generic graph traversal algorithms for scheduling actions at different nodes. Some experiments have shown that this approach is useful in some cases, but generally speaking not powerful enough. First of all we need means to detect special contexts and then act accordingly. Further more it must be possible to exclude some parts of the graph: a possible application is the exclusion of the optimisation and / or the unfolding of the body of a recursive function until that case definitely occurs.

Within the action component it should be possible to set up different transformation actions for different nodes. For instance in the context of the evaluation of a term graph expression it should be possible to switch between different evaluation orderings (eager, lazy, etc.) and try to apply only a small set of rules which is tailored towards the current term graph under transformation. The term-graph pattern approach enables the user to declare the evaluation order of some constructors as eager and that of some others as lazy. This declaration can be changed by just switching between different pattern sets. This is possible because the declaration of a brick (i. e., specification of name, arity and typing) and the transformation-strategy-related set-up for that brick are separated from each other.

A term-graph pattern is a rooted term-graph together with a sequence of actions and a continuation command. An action is a tuple consisting of a node of that term graph, two transformation expressions *pre* and *post* and a recursion statement. In order to be applicable there must be a matching homomorphism from the term graph of the pattern into the term graph under examination. An applicable term-graph pattern is applied by executing the sequence of actions. For every action the relevant node within the term graph is computed via the matching homomorphism. At first, the *pre*-expression is executed. Then the the recursion statement is checked. It controls an optional recursive call of the strategy which uses the pattern for that node. After that the *post*-expression will be evaluated. The recursive call can occur always ornever or depend on the occurence of a transformation during the *pre*-expression evaluation; we distinguish between a transformation within the term graph and a transformation which changes the relevant node. After executing one action, the transformation engine checks for the availability of another action within the sequence of the pattern. If there is no such action the application is finished. If there is another action the matching homomorphism is recomputed. If such a homomorphism does not exist any longer, the continuation command describes how to continue. The options include the termination of the application of the pattern, a reselection of another pattern from the same (i. e., the set of term-graph patterns controlled by the applied transformation strategy) or another set of patterns, specified by the pattern itself.
It is possible to have a pattern with an empty sequence of actions. This is useful in order to stop a graph traversal in a given context.

5 Example Application

In this section we shortly sketch the automatic generation of interface functions for data exchange between systems which use different data formats in a small example context.

We are interested in merging different book databases. All databases use different data formats but provide semantically equivalent data: the different formats are implementations of a common formal specification of a data model. In the concrete example a book has exactly one title, ISBN, publisher and year of publication and further on an unrestricted collection of either authors or editors.

The generation process starts with a polymorphic adaptor function. The generating transformation strategy instantiates this function and therefore produces the desired adaptor. During this instantiation the strategy distinguishes between generalised projection (extraction) cases, a change of the type of a collection and an aggregation case. The strategy is mostly controlled by type-sensitive matchings of different term-graph patterns. Altogether we use 8 polymorphic (sub-) functions, 41 transformation rules, 20 term-graph patterns and 5 (sub-) strategies.
In order to generate a new adaptor function, the user formally specifies the format of the source data by definiting a corresponding HOPS brick, inserting this brick into the generation context term graph and applying the strategy. The result of the transformation strategy application is a term graph which encodes the appropriate conversion function and does not include any polymorphic part any more.

6 Conclusion

In this extended abstract we have introduced the notion of a term-graph pattern as a powerful means for specifying transformation strategies within the Higher Object Programming System HOPS.

References

1. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
2. F. Derichsweiler. Strategy-Driven Program Transformation within the **H**igher **O**bject **P**rogramming **S**ystem HOPS. In A. Poetzsch-Heffter and J. Meyer, editors, *Programmiersprachen und Grundlagen der Programmierung*, Informatik Berichte 263 - 1/2000, pages 165–172. FernUniversität Hagen, 1999.
3. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
4. P. Hudak, S. L. Peyton Jones, P. Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.
5. N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–504, 1996.
6. W. Kahl. *Algebraische Termgraphersetzung mit gebundenen Variablen*. Reihe Informatik. Herbert Utz Verlag Wissenschaft, München, 1996. ISBN 3-931327-60-4; also Doctoral Diss. at Univ. der Bundeswehr München, Fakultät für Informatik.
7. W. Kahl. Internally typed second-order term graphs. In J. Hromkovič and O. Sýkora, editors, *Graph Theoretic Concepts in Computer Science, WG '98*, volume 1517 of *LNCS*, pages 149–163. Springer, 1998.
8. W. Kahl. The term graph programming system HOPS. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, pages 136–149, Wien, 1999. Springer-Verlag.
9. W. Kahl and F. Derichsweiler. Declarative term graph attribution for program generation. *J. UCS*, 7(1):54–70, 2001.

10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Object-Oriented Programming, 11th European Conference*, number 1241 in LNCS, 1997.
11. D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
12. W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois, 1992.
13. W. L. Scherlis. *Expression Procedures and Program Derivation.* PhD thesis, Stanford University, Department of Computer Science, August 1980.
14. C. Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research, September 1995.
15. J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. *Electronic Notes in Theoretical Computer Science*, 15, 1998.
16. E. Visser. A language for program transformation based on rewriting strategies. system description of stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications, Proc. RTA 2001*, volume 2051 of *LNCS*, pages 357–361. Springer, 2001.
17. E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:20, 1998.
18. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In P. Hudak and C. Queinnec, editors, *ICFP '98*, pages 13–26, Baltimore, MD, September 1998. ACM.
19. H. Zierer, G. Schmidt, and R. Berghammer. An interactive graphical manipulation system for higher objects based on relational algebra. In G. Tinhofer and G. Schmidt, editors, *WG '86*, volume 246 of *LNCS*, pages 68–81. Springer, 1986.

Transformational Construction of Correct Pointer Algorithms

Thorsten Ehm

Institut für Informatik
Universität Augsburg
Ehm@Informatik.Uni-Augsburg.DE

Abstract This paper shows how to use the transformation of Paterson and Hewitt (P & H) to derive imperative pointer algorithms. To achieve this we take the recursive pointer algorithms derived from functional descriptions using the method of Möller. These are transformed via the P & H transformation scheme into an imperative version. Despite the inefficient general runtime performance of the scheme that results from P & H, we get well performing algorithms.

1 Introduction

Algorithms on pointer structures are often used in lower levels of implementation. Although in modern programming languages (e.g. in Java) they are hidden from the programmer, they play a significant rôle at the implementation level due to their performance. But this advantage is bought at high expense. Pointer algorithms are very error-prone and so there is a strong demand for a formal treatment and development process for pointer algorithms. There are some approaches to achieve this goal:

Several methods [2, 11, 12] use the wp-calculus to show the correctness of pointer algorithms. There only properties of the algorithms are proved but the algorithms are not derived from a specification. So the developer has to provide an implementation. In these approaches proving trivialities may last several pages. Butler [7] investigates how to generate imperative procedures from applicative functions on abstract trees. To achieve this he enriches the trees by paths to eliminate recursion. A recent paper by Bornat [5] shows that it is possible, but difficult to reason in Hoare logic about programs that modify data structures defined by pointers. Reynolds [17] also uses Hoare logic and tries to improve a method described in a former paper of Burstall [6] to show the correctness of imperative programs that alter linked data structures.

In [14] Möller proposed a framework based on relation algebra to derive pointer algorithms from a functional specification. He shows that the rules presented also are capable of handling more difficult multi-linked data structures like doubly-linked lists or trees. However the derived algorithms are still recursive. Our goal is to improve this method by showing how to derive imperative algorithms and so achieve a more complete calculus for transformational derivation of pointer algorithms. Based on the method by Möller a recent paper by Bird [4] shows how one can derive the Schorr-Waite marking algorithm in a totally functional way.

2 Pointer structures and operations

We will give a short introduction to pointer structures and how they are used in [14]. In our model a pointer structure $\mathcal{P} = (s, P)$ consists of a store P and a list of entries

s. The entries of a pointer structure are addresses $\mathcal{A}$ that form starting points of the modeled data structures. We assume a distinguished element $\diamond \in \mathcal{A}$ representing a terminal node (e.g. null in C or nil in Pascal). A store is a family of relations (more precisely partial maps) either between addresses or from addresses to node values $\mathcal{N}_j$ such as *Integer* or *Boolean*. Each relation represents a selector on the records like e.g. *head* and *tail* for lists with functionality $\mathcal{A} \to \mathcal{N}_j$ respectively $\mathcal{A} \to \mathcal{A}$.

Each abstract object implemented is represented by a pointer structure (n, P) with a single entry $n \in \mathcal{A}$ which represents the entry point of the data structure such as for example the root node in a tree. The following operations on relations all are canonically lifted to families of relations. Algorithms on pointer structures stand out for altering links between elements. Such modification has to be modeled in the calculus as well. We use an update operator | (pronounced "onto") that overwrites relation S by relation R:

Definition 1. $R \mid S \stackrel{\text{def}}{=} R \cup \overline{dom(R)} \bowtie S$

Here we have used the *domain restriction* operator $\bowtie$ which is defined as $L \bowtie S = S \cap (L \times N)$ to select a particular part of $S \subseteq \mathcal{P}(M \times N)$. The update operator takes all links defined in R and adds the ones from S that no link starts from in R. To be able to change exactly one pointer in one explicit selector we define a sort of a "mini-store" that is a family of partial maps defined by:

Definition 2. $(x \xrightarrow{k} y) \stackrel{\text{def}}{=} \begin{Bmatrix} \{(x,y)\} & \text{for selector } k \\ \emptyset & \text{otherwise} \end{Bmatrix}$

To have a more intuitive notation leaned on traditional programming languages, we introduce the following selective update notation:

Definition 3. For selector k of type $\mathcal{A} \to \mathcal{A}$
$(n, P).k := (m, Q) \stackrel{\text{def}}{=} (n, (n \xrightarrow{k} m) \mid Q)$

which overwrites Q with a single link from n to m at selector k.

3 A running example and the problem

As example we will use a functional description of list concatenation (like e.g. (++) in Haskell [3]). We assume that the two lists are acyclic and do not share any parts. So the following pointer algorithm can be derived by transformation using the method of [14]:

$$\begin{aligned} cat_p(m, n, L) = \mathsf{if}\ m \neq \diamond\ &\mathsf{then}\ (m, L).tail := cat_p(L_{tail}(m), n, L) \\ &\mathsf{else}\ (n, L) \end{aligned}$$

The two pointer structures (m, L) and (n, L) are representations of the two lists. Addresses m and n model the starting points, whereas L is the memory going with them. In other words m and n form links to the beginning of two lists in memory L.

Note that this is only one candidate of possible implementations for the functionally described specification of (++). Because we are interested in algorithms performing minimal **destructive** updates we did not derive a persistent variant such as the standard, partially copying interpretation in functional languages.

We now have a linear recursive function working on pointer structures. But what we want is an imperative program that does not use recursion. By investigating the execution order of cat_p we can see, that cat_p calculates a term of the following form:

$$(m, L).tail := ((L_{tail}(m), L).tail := ...(n, L))$$

If you remember the definition of the := operator, this means that updates are performed from right to left.

$$(m \overset{tail}{\to} L_{tail}(m)) \mid (\ldots \mid ((L^k_{tail}(m) \overset{tail}{\to} n) \mid L) \ldots)$$

This shows that the derived algorithm uses the update operator not only to properly alter links but also to just pass through the structure after returning from the recursion.

As we can see, there are several such updates that do not alter the pointer structure. For example $(m \overset{tail}{\to} L_{tail}(m))$ is already contained in L and does not change the pointer structure $(\ldots \mid ((L^k_{tail}(m) \overset{tail}{\to} n) \mid L) \ldots)$ if the previous updates do not affect this part of L. This is the case for several algorithms on pointer linked data structures, because most of them first have to scan the structure to find the position where they have to do the proper changes.

In transformational program design the transformation of a linearly recursive function to an imperative version always has two steps: First transform the linear recursion into tail recursion. Then apply a standard transformation scheme [16] to get a while program. But cat_p does not have tail recursive form. So we first have to find a way to transform cat_p into the right form. There are several schemes to derive a tail recursive variant from a linear recursive function [1]. But the function $K(m, n, L)$ is not good-natured enough to be able to apply one of these standard methods. So is there no way to get a tail recursive version of cat_p ?

4 The transformation scheme of Paterson/Hewitt

In 1970 Paterson and Hewitt presented a transformation scheme that makes it possible to transform any linear recursive function to a tail recursive one [1]. This rule normally is only of theoretical interest because of the bad runtime performance of the resulting function. P & H applied the idea of using the inverse function $\overline{K}$ to make the step from K^{i+1} to K^i, but exhaustively recalculated K^i from the start. The evolving scheme is:

$$F(x) = \textsf{if } B(x) \textsf{ then } \phi(F(K(x)), E(x)) \textsf{ else } H(x)$$

$$\Updownarrow \quad [\text{ P \& H}$$

$$\begin{aligned}
F(x) = {} & G(n0, H(m0)) \text{ where} \\
& (m0, n0) = num(x, 0) \\
& num(y, i) = \textsf{if } B(y) \textsf{ then } num(K(y), i+1) \\
& \qquad\qquad\qquad \textsf{else } (y, i) \\
& it(y, i) = \textsf{if } i \neq 0 \textsf{ then } it(K(y), i-1) \\
& \qquad\qquad\quad \textsf{else } y \\
& G(i, z) = \textsf{if } i \neq 0 \textsf{ then } G(i-1, \phi(z, E(it(x, i-1)))) \\
& \qquad\qquad\quad \textsf{else } z
\end{aligned}$$

The function num calculates the number of iterations that have to be done until the termination condition is fulfilled as well as the final value. These values are used by function G to change the evaluation order of the calculated term. For this, G uses the function it to iterate K to achieve the inverse $\overline{K}$ of K by doing one iteration less than had to be done for K. So G can start with the calculations done in the deepest recursion step first and then ascend from there using the inverse of K.

5 Deriving a general transformation scheme

By investigation of function $\phi_k((m, L), (n, L)) = (n, (n \overset{k}{\rightarrow} m) \mid L)$ we can see that ϕ_k updates the link starting from m via selector k and simultaneously sets m as the new starting entry of the resulting pointer structure. It is apparent that such a restricted function can not provide the simplification we aim to achieve, namely elimination of effect-less updates. So we use the technique of generalization and introduce a more flexible function $\psi_k(l, m, (n, L)) = (l, (m \overset{k}{\rightarrow} n) \mid L)$ that handles the altered address and the resulting entry independently. With this function we are in the position to eliminate the quasi-updates that do not alter the structure but are only used for passing through the pointer structure and get a non-recursive function G. One can say that ψ_k "eats up" the effect-less updates of ϕ_k. The scheme that evolves from some calculations is:

$$
\boxed{
\begin{array}{l}
F(x) = \textsf{if } B(x) \ \textsf{ then } \phi(F(K(x)), E(x)) \\
\qquad\qquad\qquad\ \ \textsf{else } H(x) \\
\hline
\quad\updownarrow \qquad\qquad\qquad\qquad\qquad\qquad [\ \text{Conditions} \\
\hline
F(x) = \textsf{var } vx := x \\
\qquad\quad \textsf{if } B(x) \ \textsf{ then } \ \textsf{while } B(K(vx)) \textsf{ do } vx := K(vx) \\
\qquad\qquad\qquad\qquad\quad \psi_k(ptr(E(x)), ptr(E(vx)), H(K(vx))) \\
\qquad\qquad\qquad\ \ \textsf{else } H(x)
\end{array}
}
$$

Some more simplification leads us to the imperative algorithm one has in mind:

$$
\begin{array}{l}
cat_p(m, n, L) = \textsf{var } vm := m \\
\qquad\qquad\qquad\quad \textsf{if } m \neq \diamond \ \textsf{ then } \ \textsf{while } L_{tail}(vm) \neq \diamond \textsf{ do } vm := L_{tail}(vm) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (m, (vm \overset{tail}{\rightarrow} n) \mid L) \\
\qquad\qquad\qquad\qquad\qquad\ \ \textsf{else} \quad (n, L)
\end{array}
$$

6 Conclusion

We have shown how the transformation of Paterson and Hewitt can be used to achieve imperative algorithms on pointer-linked data structures. The presented transformation scheme also can be applied to other algorithms like insert into a list or tree [9]. At these example algorithms it can be seen, that there is a need for more sophisticated schemes based on the presented one. It also seems possible that algorithms changing more than one link such as deletion from a list can be treated the same way. For this, one have to divide the job into several parts altering only one link, applying the scheme and afterwards putting the parts together.

Further research will investigate this and other starting points to complete the methodology. Also a (semi-)automatic system checking the side-conditions and so supporting the developer of such algorithms is in work.

References

1. F.L. Bauer, H. Wössner: *Algorithmic Language and Program Development*, Springer, Berlin, 1984
2. A. Bijlsma: *Calculating with pointers.* Science of Computer Programming **12**, Elsevier 1989, 191–205
3. R. Bird: *Introduction to Functional Programming using Haskell*, 2nd edition, Prentice Hall Press, 1998
4. R. Bird: *Unfolding Pointer Algoritms*, under consideration for publication in Journal of Functional Programming, available from: `http://www.comlab.ox.ac.uk/oucl/work/richard.bird/publications`
5. R.Bornat: *Proving pointer programs in Hoare logic.* Proceedings of MPC 2000, Ponte de Lima , LNCS 1837, Springer 2000, 102–126
6. R. Burstall: *Some techniques for proving correctness of programs which alter data structures.* In B. Meltzer and D. Michie eds, Machine intelligence 7, Edinburgh University Press, 1972, 23–50
7. M. Butler: *Calculational derivation of pointer algorithms from tree operations.* Science of Computer Programming **33**, Elsevier 1999, 221–260
8. T. Ehm: *Case studies for the derivation of pointer algorithms.* to appear
9. T. Ehm: *Transformational construction of correct pointer algorithms.* Proceedings of PSI 2001, Novosibirsk, Springer LNCS, to appear.
10. C. A. R. Hoare: *Proofs of correctness of data representations.* Acta Informatica **1**, 1972, 271–281
11. J.M. Morris: *A general axiom of assignment.* Theoretical Foundations of Programming Methodology, NATO Advanced Study Institutes Series C Mathematical and Physical Sciences **91**, Dordrecht, Reidel, 1981, 25–34
12. J.M. Morris: *Assignment and linked data structures.* Theoretical Foundations of Programming Methodology, NATO Advanced Study Institutes Series C Mathematical and Physical Sciences **91**, Dordrecht, Reidel, 1981, 35–51
13. B. Möller: *Towards pointer algebra.* Science of Computer Programming **21**, Elsevier, 1993, 57–90
14. B. Möller: *Calculating with pointer structures.* In: R. Bird, L. Meertens (eds.): Algorithmic languages and calculi. Proc. IFIP TC2/WG2.1 Working conference, Le Bischenberg, Feb. 1997. Chapman & Hall 1997, 24–48
15. B. Möller: *Calculating with acyclic and cyclic lists.* In A. Jaoua, G. Schmidt (eds.): Relational Methods in Computer Science. Special Issue on the 3rd Int. Seminar on Relational Methods in Computer Science, Jan 6–10, 1997 in Hammamet Tunisia. Information Sciences — An International Journal **119**, 1999, 135–154
16. H. Partsch: *Specification and transformation of programs. A formal approach to software development.* Monographs in Computer Science. Springer, 1990
17. J.C.Reynolds: *Intuitionistic reasoning about shared mutable data structures.* In: Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare, Palgrave, 2000

Dynamic Scope Analysis for Emacs Lisp

Matthias Neubauer

Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee 079
79110 Freiburg i. Br.

Abstract It is possible to translate code written in Emacs Lisp or another Lisp dialect which uses dynamic scoping to a more modern programming language with lexical scoping while largely preserving structure and readability of the code. The biggest obstacle to such an idiomatic translation from Emacs Lisp is the translation of dynamic binding into suitable instances of lexical binding: Many binding constructs in real programs in fact exhibit identical behavior under both dynamic and lexical binding. An idiomatic translation needs to detect as many of these binding constructs as possible and convert them into lexical binding constructs in the target language to achieve readability and efficiency of the target code.

The basic prerequisite for such an idiomatic translation is thus a *dynamic scope analysis* which associates variable occurrences with binding constructs. We present such an analysis. It is an application of the Nielson/Nielson framework for flow analysis to a semantics for dynamic binding akin to Moreau's. Its implementation handles a substantial portion of Emacs Lisp, has been applied to realistic Emacs Lisp code, and is highly accurate and reasonably efficient in practice.

1 Migrating Emacs Lisp

Emacs Lisp [2, 8] is a popular programming language for a considerable number of desktop applications which run within the Emacs editor or one of its variants. The actively maintained code base measures at around 1,000,000 loc[1]. As the Emacs Lisp code base is growing, the language is showing its age: It lacks important concepts from modern functional programming practice as well as provisions for large-scale modularity. Its implementations are slow compared to mainstream implementations of other Lisp dialects. Moreover, the development of both Emacs dialects places comparatively little focus on significant improvements of the Emacs Lisp interpreter.

On the other hand, recent years have seen the advent of a large number of *extension language* implementations of full programming languages suitable for the inclusion in application software. Specifically, several current Scheme implementations are technologically much better suited as an extension language for Emacs than Emacs Lisp itself. In fact, the official long-range plan for GNU Emacs is to replace the Emacs Lisp substrate with Guile, also a Scheme implementation [7]. The work presented here is part of a different, independent effort to do the same for XEmacs, a variant of GNU Emacs which also uses Emacs Lisp as its extension language.

Replacing such a central part of an application like XEmacs presents difficult pragmatic problems: It is not feasible to re-implement the entire Emacs Lisp code base by hand. Thus, a successful migration requires at least the following ingredients:

[1] The XEmacs package collection which includes many popular add-ons and applications currently contains more than 700,000 loc.

- Emacs Lisp code must continue to run unchanged for a transitory period.
- An automatic tool translates Emacs Lisp code into the language of the new substrate, and it must produce maintainable code.

Whereas the first of these ingredients is not particularly hard to implement (either by keeping the old Emacs Lisp implementation around or by re-implementing an Emacs Lisp engine in the new substrate), the second is more difficult. Even though a direct one-to-one translation of Emacs Lisp into a modern latently-typed functional language is straightforward by using dynamic assignment or dynamic-environment passing to implement dynamic scoping, it does not result in maintainable output code: Users of modern functional languages use dynamic binding only in very limited contexts such as exception handling or parameterization. As it turns out, the situation is not much different for Emacs Lisp users: For many `let`s and other binding constructs in real Emacs Lisp code, dynamic scope and lexical scope are *identical*! Consequently, a good "idiomatic" translation of Emacs Lisp into, say, Scheme, should convert these binding constructs into the corresponding lexical binding constructs of the target substrate.

The only problem is to *recognize* these binding constructs, or rather, distinguish those where the programmer "meant" dynamic scope from those where she "meant" lexical scope. Since with dynamic scope, bindings travel through the program execution much as values do, this requires a proper flow analysis. We present such an analysis that we call *dynamic scope analysis*.

Specifically, our contributions are the following:

- We have formulated a semantics for a subset of Emacs Lisp, called Mini Emacs Lisp, similar to the *sequential evaluation function* for Λ_d by Moreau [3].
- We have applied the flow analysis framework of Nielson and Nielson [6] to the semantics, resulting in an acceptability relation for flow analyses of Mini Emacs Lisp programs.
- We have used the acceptability relation to formulate and implement a flow analysis for Emacs Lisp which tracks the flow of bindings in addition to the flow of values.
- We have applied the analysis to real Emacs Lisp code. More specifically, the analysis is able to handle medium"=sized real-world examples with high accuracy and reasonable efficiency.

The work presented here is a part of the `el2scm` project that works on the migration from Emacs Lisp to Scheme. However, the analysis could be used for a number of other purposes, among them the development of an efficient compiler for Emacs Lisp, or the translation to a different substrate such as Common Lisp. For further technical details of the dynamic scope analysis, the reader is referred to the author's thesis dissertation [4] and to [5].

2 Examples

Consider the Emacs Lisp code shown in Figure 1, taken literally from `files.el` in the current XEmacs core. It contains five variable bindings, all introducing temporary names for intermediate values. The bindings of the variables `filename`, `file`, `dir`, `comp`, and `newest` are all visible in the other functions reachable from the body

```
(let* ((filename (expand-file-name filename))
       (file (file-name-nondirectory filename))
       (dir  (file-name-directory    filename))
       (comp (file-name-all-completions file dir))
       newest)
  (while comp
    (setq file (concat dir (car comp))
          comp (cdr comp))
    (if (and (backup-file-name-p file)
             (or (null newest)
                 (file-newer-than-file-p file newest)))
        (setq newest file)))
  newest))
```

Figure 1. Typical usage of `let` in Emacs Lisp.

```
(let ((file-name-handler-alist nil)
      (format-alist nil)
      (after-insert-file-functions nil)
      (coding-system-for-read 'binary)
      (coding-system-for-write 'binary)
      (find-buffer-file-type-function
       (if (fboundp 'find-buffer-file-type)
           (symbol-function 'find-buffer-file-type)
         nil)))
  (unwind-protect
      (progn
        (fset 'find-buffer-file-type
              (lambda (filename) t))
        (insert-file-contents
          filename visit start end replace))
    (if find-buffer-file-type-function
        (fset 'find-buffer-file-type
              find-buffer-file-type-function)
      (fmakunbound 'find-buffer-file-type))))
```

Figure 2. Parameterizations via dynamic `let` in Emacs Lisp.

of the `let`, yet none of them contain occurrences of these names. The only variable occurrences which access the bindings are in the body of the `let*` itself, and all are within the lexical scope of the bindings. Hence, translating the `let*` into a lexically-scoped counterpart in the target language would preserve the behavior of this function.

Figure 2 shows an example for idiomatic use of dynamic binding (also taken from `files.el`): It is part of the implementation of `insert-file-contents-literally` which calls `insert-file-contents` in the body of the `let`. The definition of `insert-file-contents` indeed contains occurrences of the variables bound in the `let` with the exception of `find-buffer-file-type-function`. Therefore, it is not permissible to translate the `let` with a lexically-scoped binding construct.

For the vast majority of binding constructs in real Emacs Lisp code, dynamic scope and lexical scope coincide. Thus, the ultimate goal of the analysis is to detect as many of these bindings constructs as possible.

In general however, value flow and the flow of bindings interact during the evaluation of Emacs Lisp programs. Hence, it is not possible to apply standard flow analyses based on lexical-binding semantics to solve the problem; a new analysis is necessary.

3 Conclusion and Future Work

We have specified, proved correct and implemented a flow analysis for Emacs Lisp whose distinguishing feature is its correct handling of dynamic binding. The primary purpose of the analysis is to aid translation of Emacs Lisp programs into more modern language substrates with lexical scoping since most binding in real Emacs Lisp programs behaves identically under lexical and dynamic scoping. Our analysis is highly accurate in practice. Our prototype implementation is reasonably efficient.

We have two main directions for future research:

- Improving the efficiency of the analysis by ordinary optimization, compilation code and modularization of the constraints [1], and
- integration of the analysis into a translation suite from Emacs Lisp to Scheme.

References

1. Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, March 1999.
2. Bil Lewis, Dan LaLiberte, Richard Stallman, and the GNU Manual Group. GNU Emacs Lisp reference manual. `http://www.gnu.org/manual/elisp-manual-20-2.5/elisp.html`, 1785.
3. Luc Moreau. A Syntactic Theory of Dynamic Binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, December 1998.
4. Matthias Neubauer. Dynamic scope analysis for Emacs Lisp. Master's thesis, Eberhard-Karls-Universität Tübingen, December 2000. `http://www.informatik.uni-freiburg.de/~neubauer/diplom.ps.gz`.
5. Matthias Neubauer and Michael Sperber. Down with emacs lisp: Dynamic scope analysis. In Xavier Leroy, editor, *Proc. International Conference on Functional Programming 2001*, pages 38–49, Florence, Italy, September 2001. ACM Press, New York.
6. Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. POPL'97*, pages 332–345. ACM Press, 1997.
7. Richard Stallman. GNU extension language plans. Usenet article, October 1994.
8. Ben Wing. XEmacs Lisp Reference Manual. `ftp://ftp.xemacs.org/pub/xemacs/docs/a4/lispref-a4.pdf.gz`, May 1999. Version 3.4.

Konsistenz von Vererbung in objektorientierten Sprachen und von statischer, ALGOL-artiger Bindung

Hans Langmaack

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel

Mit der höheren, problemorientierten Programmiersprache ALGOL60 [19] wurde etwas softwaretechnisch sehr Wichtiges geschaffen, nämlich das Blockkonzept. K. Samelson brachte es in ALGOL60 ein, die Niveaustruktur von Blöcken und die damit einhergehende Speicherverteilung gehören zu den bedeutendsten Beiträgen Samelsons zur Programmierungstechnik. Im ALGOL58-Bericht [21] und in FORTRAN [1] war noch keine Rede von Bindungs- und Gültigkeits-(Sichtbarkeits-) bereichen von Identifikatoren, Phänomenen, die in Prädikatenlogik und λ-Kalkülen seit den 1930er Jahren bekannt waren [8].

Erst das Blockkonzept brachte tragfähige Klärung in den Prozedurbegriff, syntaktisch wie semantisch. Die ALGOL60-Prozeduren sind aus den do-Anweisungen und Prozeduren von ALGOL58 hervorgegangen. Das ALGOL-artige, statische Binden von Identifikatoren wurde zudem dadurch zum Ausdruck gebracht, daß bei Prozedurrumpf- und Parameterersetzung (operationelle Kopierregelsemantik) Bindungsverfälschungen durch gebundene Identifikatorumbenennungen zu vermeiden waren. Die im ALGOL60-Bericht verwendeten Formulierungen zur Sprachsemantik, speziell zur Semantik von Funktions- und eigentlichen Prozeduren, wären für Programmierer und Übersetzerkonstrukteure erheblich klarer geworden, wenn der Bericht explizit auf die damals schon bekannte α- und β-Reduktion in λ-Kalkülen hingewiesen hätte. Das hätte manche unglückliche Programmiersprachen-, Übersetzer- und Laufzeitsystementwicklung vermeiden helfen.

Die Idee der Blöcke und ihr Datenspeicherverhalten hatte Samelson schon in [24] vorgezeichnet. Er sprach von Teilprogrammen als offenen Unterprogrammen und Bibliotheksprogrammen als geschlossenen Unterprogrammen. Er beschrieb einerseits, wie beim Übersetzen und Auswerten arithmetischer Formeln Zwischenergebnisse in Hilfsspeicherzellen unterzubringen waren, wobei zuletzt besetzte zuerst wieder verfügbar wurden. Das lag am Formelabbau von links nach rechts, es wurde immer wieder jeweils die vorderste Abbaumöglichkeit ins Auge gefaßt, ein berechtigtes Vorgehen, weil das Auswerten arithmetischer Formeln, auch der schulbekannten in Infixnotation mit den Klammereinsparungs- und Vorrangregeln, konfluent ist. Die Bezeichnung Zahlkeller für die pulsierend auftretenden Zwischenergebnisse trat erst in der Patentschrift [2] und öffentlich wirksam im Artikel "Sequentielle Formelübersetzung" [23] auf. Andererseits beschrieb Samelson 1955 aber auch, wie sich das Pulsieren der Zwischenergebnisse zur Laufzeit auf den Datenspeicher für Teil- und Unterprogramme ausdehnen ließ, berechtigt ebenfalls aus Konfluenzgründen. Teil- und Unterprogramme hatten mit einer Angabe des für Rechnungen jeweils freien Speichers zu arbeiten, eingetragen unter einem festen Variablennamen *Anfang freier Speicher*.

Unter ausdrücklicher Berufung auf [23] beschrieb Dijkstra in "Recursive Programming" [6], wie der Samelson-Bauersche Zahlkeller zum Laufzeitkeller (*run time stack*) für Blöcke, Funktions- und eigentliche Prozeduren der Sprache ALGOL60 auszudehnen war. Die Variable *Anfang freier Speicher* hieß da *stack pointer*; jeder Prozeduraufruf erzeugte eine *Prozedurinkarnation*, für die eine *Informationseinheit* mit Plätzen für Koordination (*link*), für lokale Parameter, lokale Variablen und Zwischenresultate in den Laufzeitkeller eingetragen wurde. Im Prozedur-*link* etablierte Dijkstra neben der Rückkehradresse und dem *dynamischen Zeiger* auf die letzte zeitlich vorangegangene, aber noch nicht beendete Inkarnation neu den sog. *zweiten Parameterzeiger*, den *statischen Zeiger* auf die *jüngste* (*most recent*) Inkarnation der lexikographisch umfassenden Prozedur, um über ihn an Informationen zu globalen Prozedurparametern zu gelangen.

Leider geht Dijkstra's "*most recent*"-Festlegung nicht mit der Kopierregel, der statischen Bindung des ALGOL60-Berichts konform. Denn man kann Programme konstruieren, die nicht die sog. "*most recent*"-Eigenschaft haben [7]. Bei Dijkstras Implementierung erfahren Identifikatoren während des Ausführungsprozesses u. U. unvermutete Bedeutungsänderungen, die der Programmierer kaum nachvollziehen kann, so daß er von seltsamen Endergebnissen überrascht wird. Man spricht von Sprachsemantik mit dynamischer Bindung, wenn derartige Bedeutungsänderungen gewollt sind. Selbst das aufwendige Programmbeispiel GPS (General Problem Solver) in [22] zur Demonstration von Namensparameterübergabe erfüllt die "*most recent*"-Eigenschaft.

Dijkstras "*most recent*"-Vorschrift zur Behandlung des statischen Zeigers fand selbst noch in jüngerer Zeit Eingang in Implementierungen und Übersetzerbaulehrbücher für ALGOL-artige Programmiersprachen. Das führte natürlich zu Mißhelligkeiten zwischen ursprünglicher Sprachsemantik und aktueller Programmausführung. Um Enttäuschungen aus dem Weg zu gehen, wurden beispielsweise Prozeduren als Argumente von Prozeduren und im Gefolge formale Proteduraufrufe in Ada [10] und Prozedurschachtelungen in C [12] nicht mehr erlaubt. Tatsächlich stimmt unter diesen Spracheinschränkungen statische und dynamische Bindung in ihren Auswirkungen überein.

Das dynamische Binden wurde ungewollt durch eine weitere einflußreiche Veröffentlichung ins Programmiererbewußtsein gerückt. 1965 veröffentlichten J. McCarthy et al. im "Lisp 1.5 Programmer's Manual" [17] zwei in Lisp geschriebene Interpretierer zur Definition operationeller Kopierregelsemantik der funktionalen Sprache Lisp. McCarthy hatte Lisp als benutzerfreundliche Fassung des auf Church zurückgehenden (angewandten) λ-Kalküls mit dessen statischer Bindung konzipiert [9]. Aber Programmierfehler in den Interpretierern führten zu Lisp-Semantik mit dynamischer Bindung, welche bei bloßer gebundener Umbenennung schon unterschiedliche Programmresultate zeitigte. Langmaack entdeckte die Programmierfehler in [17] im Rahmen von Vorlesungen zu Übersetzerbau und Laufzeitsystemen an der Universität des Saarlandes 1970/71. Er besserte die Lisp-Interpretierer kurzerhand in Richtung statischer Bindung aus und sprach von natürlicher Semantik. Erst später kehrte die Lisp-Gemeinde mit CommonLisp [25] zur Tugend ALGOL-artiger, statischer Bindung zurück.

Man sollte sich durch die Wortwahl "dynamisches Binden" nicht dahingehend täuschen lassen, daß diese Form des Bindens etwa klaren Vorteil oder hohe Mächtigkeit biete. Denn solches Binden verlangt zur Besetzung statischer Zeiger Suchprozesse im Laufzeitkeller, während statisches Binden gezieltes Besetzen gestattet. Auch die bei dynamischem Binden beweisbare Existenz relativ vollständiger Hoarescher Beweissysteme [20] muß mit schwierigerem Programmverstehen und umständlicher formulierbaren Prozedurvor- und -nachinvarianten erkauft werden, weil keine glatten Substitutions- und Umbenennungstheoreme wie bei statischer Semantik gelten.

Das dynamische Binden wurde vor allem in objektorientierter Programmierung populär, obwohl O.-J. Dahl und K. Nygaard, die Schöpfer der Begriffe Objekt, Klasse, Vererbung und der Sprache Simula67, ausdrücklich auf ALGOL60 mit der Blockstruktur und dem statischen Binden fußten [3, 5]. Auch die Simula67-Nachfolgesprachen BETA [18] und LOGLAN'88 [16] verbinden Objektorientierung konsistent und erfolgreich mit statischer Bindung. Während unabhängig entwickelte Programmteile etwa durch Prozedurkapselung schon in ALGOL60 oder PASCAL [11] problemlos kombiniert werden können, ist solch softwaretechnisches Vorgehen bei dynamischer Bindung ohne Kenntnis der lokalen Namen in anderen Programmteilen fehleranfällig oder sogar schon aus syntaktischen Gründen unmöglich. Wie gesagt, Simula67 beinhaltete bereits die Vererbungsidee, verlangte aber aus pragmatischen Gründen Einebenenvererbung, d.h. ererbte Klassen mußten gleiche Modulschachtelungstiefen wie erbende Moduln haben. Für LOGLAN wie für BETA wurde dagegen Mehrebenenvererbung angestrebt, u.a. um Programmierer nicht zu zwingen, unnötige Klassenkopien per Hand zu schreiben, und um flexiblere Einrichtung von Klassenbibliotheken zu ermöglichen. Daß es nicht einfach sein würde, für Mehrebenenvererbung klare Sprachsemantik mit statischer Bindung zu definieren und effiziente Implementierungen zu erreichen, wurde 1967 noch nicht vorausgesehen [4]. Für LOGLAN'88 läßt sich operationelle Kopierregelsemantik mit statischer Bindung wie für ALGOL-artige Sprachen definieren. Nicht nur Prozeduraufrufe, Funktionsaufrufe und Objektgenerierungen erwarten Kopierregeln, sondern auch das Eliminieren von Ererbungen tut es, womit Module wie Klassen, Blöcke, eigentliche Prozeduren und Funktionsprozeduren versehen sein können. Dieser Definitionsstil verbleibt voll und ganz auf Programmiersprachebene, ohne Bezug zu irgendeiner Implementierung oder Maschine zu haben [13].

Für effiziente Implementierung ist die Idee wegweisend, daß man in semantikerhaltender Weise das Ererben auch dadurch eliminieren kann, daß man Klassen in Prozeduren verwandelt und erbende Moduln mit neuen lokalen Prozeduren versieht, deren formale Parameter gerade diejenigen Identifikatoren sind, die über die Erbungs-(Präfix-)kette des erbenden Moduls erreichbar sind. M.a.W.: Objektorientierte Programme mit Klassen und Vererbung sind angenehm verkürzende und parametersparende Notation für spezielle ALGOL-Programme.

Effiziente ALGOL-artige Block- und Prozedurimplementierung erreicht man gemäß [6] durch statische Zeigerketten und Displayregister. Dabei sind alle angewandten Identifikatorvorkommen, die das gleiche zugehörige definierende Vorkommen haben und somit bei statischer Bindung semantisch das gleiche Ding bedeuten, an das gleiche Register gekoppelt, festgelegt durch das Modulschachtelungsniveau des definierenden Vorkommens. Wegen der Einebenenvererbung bleibt das auch für Simula67 richtig, mit der angenehmen Folge, daß keinerlei Diplayregisterumladungen erforderlich werden, solange ein Rechnen der Laufzeit in einer Präfixkette verharrt und diese nicht verläßt.

Die ALGOL-Simula67-Displayregisterverteilung wird falsch bei Mehrebenenvererbung. Aber die letztgenannte Eigenschaft einer Simula67-Implementierung, Displayregister nicht umladen zu müssen, möchte man im Interesse effektvollen objektorientierten Programmierens bei Mehrebenenvererbung bewahren. Dadurch werden u.U. mehr Displayregister nötig, als das maximale Modulschachtelungsniveau eines Programms angibt. Krogdahl [15] schlug daher vor, den BETA-Codegenerator so zu optimieren, daß er zur Übersetzungszeit die niedrigst mögliche Displayregisterzahl bestimmt. [13] beweist durch systematische Displayregisterpermutation, daß diese Zahl grundsätzlich gerade durch die maximale Modulschachtelungstiefe gegeben ist. Kreczmar und Warpechowski [14] entwickelten dazu eine elegante Theorie statischer und dynamischer Algebren, wofür LOGLAN'88-Programme Modelle sind. Theorie und Implementierung sind Ergebnisse von Überlegungen darüber, was Programmiersprachsemantik mit statischer Bindung eigentlich bedeutet.

Ein- und Mehrebenenvererbung sollten nicht mit den Phänomenen einfache bzw. multiple Vererbung verwechselt werden. Alle drei Sprachen Simula67, BETA und LOGLAN'88 erlauben nur einfache Vererbung. Es steht noch aus, wie multiple Vererbung in natürlicher Weise mit statischer Bindung gekoppelt werden kann.

Meiner Kollegin G. Mirkowska und meinen Kollegen O.-J. Dahl, C.A.R. Hoare und A. Salwicki danke ich herzlich für die Diskussionen um Objektorientierung und statisches Binden.

Literatur

1. J.W. Backus et al.. The FORTRAN Automatic Coding System. Proc. Western Joint Computing Conf. 11, 188-198, 1957
2. F.L. Bauer, K. Samelson. Verfahren zu automatischen Verarbeitung von kodierten Daten und Rechenmaschinen zur Ausübung des Verfahrens. Patentanmeldung Deutsches Patentamt, 1957
3. O.-J. Dahl. The Birth of Object Orientation: The Simula Languages. Lecture, sd & m Conf. on Software Pioneers, 11 pp, Bonn 2001
4. O.-J. Dahl. Persönliche Korrespondenz. Asker 2001
5. O.-J. Dahl, K. Nygaard. Class and Subclass Declarations. In: J.N.Buxton (ed.). Simulation Programming Languages. Proc. IFIP Work. Conf. Oslo 1967, North Holland, Amsterdam, 158-174, 1968
6. E.W. Dijkstra. Recursive Programming. Num. Math. 2, 312-318, 1960
7. A.A. Grau, U. Hill, H. Langmaack. Translation of ALGOL60. Handbook for Automatic Computation Ib, chief ed. K.Samelson. Springer-Verlag, Berlin, Heidelberg, New York 1967
8. H. Hermes. Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit. Springer-Verlag, Berlin, Göttingen, Heidelberg 1961
9. C.A.R. Hoare. Persönliche Mitteilung. EU-Projekt "Provably Correct Systems - ProCoS", Oxford 1990, Cambridge 2001
10. J.D. Ichbiah. Ada Reference Manual. LNCS 106, Springer-Verlag, Berlin, Heidelberg, New York 1980
11. K. Jensen, N. Wirth. PASCAL-User Manual and Report, 2nd ed.. Springer Verlag, New York, Heidelberg, Berlin 1975
12. B.W. Kernighan, D.M. Ritchie. The C Programming Language. Prentice Hall, Englewood Cliffs N.Y. 1978
13. A. Kreczmar, M. Krause, A. Salwicki, H. Langmaack. Specification and Implementation Problems of Programming Languages Proper for Hierarchical Data Types. Bericht 8410, Inst.Informatik Prakt. Math. CAU Kiel, 1984
14. M. Krause, H. Langmaack, A. Kreczmar, M. Warpechowski. Concatenation of Program Modules, an Algebraic Approach to Semantic and Implementation Problems. In: A. Skowron (ed.). Computation Theory. Proc. 5th Symp. Zaborow 1984, LNCS 208, Springer Verlag, Berlin, Heidelberg, New York, 134-156, 1985

15. S. Krogdahl. On the Implementation of BETA. Norwegian Comp. Centre, 1979
16. A. Kreczmar, A. Salwicki, M. Warpechowski. LOGLAN'88-Report on the Programming Language. LNCS 414, Springer-Verlag, Berlin, Heidelberg, New York 1990
17. J. McCarthy et al.. Lisp 1.5 Programmer's Manual. The M.I.T. Press, Cambridge Mas. 1965
18. O.L. Madsen, B. Møller-Pedersen, K. Nygaard. Object Oriented Programming in the BETA Programming Language. Addison Wesley / ACM Press, 1993
19. P. Naur (ed.) et al.. Report on the Algorithmic Language ALGOL60. Num. Math. 2, 106-136, 1960
20. E.-R. Olderog. Sound and complete Hoare-like calculi based on copy rules. Acta Informatica, 16, 161-197, 1981
21. ACM Committee on Programming Languages and GAMM Committee on Programming, ed. by A.J. Perlis, K. Samelson. Report on the Algorithmic Language ALGOL. Num. Math. 1, 41-60, 1959
22. B. Randell, L.J. Russell. ALGOL60 Implementation. Academic Press, London, New York 1964
23. K. Samelson, F.L. Bauer. Sequentielle Formelübersetzung. Elektr. Rechenanl. 1, 4, 176-182, 1959
24. K. Samelson. Probleme der Programmierungstechnik. Intern. Koll. über Probleme der Rechentechnik, Dresden 1955, VEB Deutscher Verlag der Wissenschaften, Berlin, 61-68, 1957
25. G.L. Steele jr.. CommonLisp: The Language. Digital Press, 1984

Object-Oriented Specification and Verification with Co-Algebras and Co-Induction

Peter Padawitz

University of Dortmund, Germany

1 Swinging types

Swinging types (STs) provide a specification and verification formalism for designing software in terms of many-sorted logic. Current formalisms, be they set- or order-theoretic, algebraic or coalgebraic, rule- or net-based, handle either static system components (in terms of functions or relations) or dynamic ones (in terms of transition systems) and either structural or behavioral aspects, while STs combine equational, Horn and modal logic for the purpose of applying computation and proof rules from all three logics.

UML provides a collection of object-oriented pictorial specification techniques, equipped with an informal semantics, but hardly cares about consistency, i.e. the guarantee that a specification has models and thus can be implemented. To achieve this goal and to make verification possible a formal semantics is indispensable. Swinging types have term models that are directly derived from the specifications. We take first steps towards a translation of class diagrams, OCL constraints and state machines into STs.

Swinging types are particularly suitable for interpreting UML models because they integrate static and dynamic components. UML treats them separately, STs handle them within the same formalism. Hence, one may prove, for instance, that static operations are correctly refined by local message passing primitives.

A crucial point of a formal semantics of UML models is a reasonable notion of *state*. If constraints are considered that involve static data *and* state transitions, the modal-logic representation of states as (implicit) predicates is often less adequate than the ST representation as terms even if a state may have several term representations, which denote, for instance, different method sequences that lead to the state.

Given a system to be analyzed or synthesized, there are two conceptionally and technically rather different views on the relationship between its static structure on the one hand and its dynamic behavior on the other: the **two-tiered view** and the **one-tiered view**, respectively. The former is based on temporal and modal logic where each state has its own interpretation ("world") of all syntactic entities that build up the system. Here the formal semantics is given by a *Kripke structure*, i.e. a sequence of models each of which describes a single state. The state structure does not interfere with the transition relation that captures the dynamics and only the dynamics of the system.

In contrast to the two-tiered view, formal approaches adopting the one-tiered view regard states not as different models, but as different elements of a single model. This allows us to keep to—many-sorted—predicate logic: *hidden* domains of states are distinguished from *visible* domains such as numbers or finite lists. Visible domains consist of data that are identified by their structure, while the objects of hidden domains are identified by their behavior in response to *observers*. Approaches that favor a one-tiered view are process algebra [4], dynamic data types [2], hidden algebras [8, 10] and swinging types. Hidden algebras are closely related to models of behavioral specifications [3] and subsumed by models of coalgebra specifications [12, 13, 25, 29]. Hidden-algebra specifications axiomatize behavioral equivalence; dynamic data types specify transition systems; swinging types do both.

Consequently, a swinging type separates visible sorts from hidden ones. ST predicates are interpreted as the least relations satisfying their axioms and thus represent inductive(ly provable) properties. Dually, *copredicates* are interpreted as the greatest relations satisfying their axioms and often represent complementary, "coinductive" properties.[1] A *coalgebraic ST* [21] includes the specification of a *final coalgebra*, which can be regarded as a "contracted" Kripke structure: each state model of a Kripke structure corresponds to an *element* of the coalgebra.

Since the semantics of an ST is given by a Herbrand structure, swinging types combine features of model-oriented formal description techniques with those of axiomatic, deduction-oriented ones. This is quite natural, puts powerful proof rules at our disposal, like unfolding, induction, coinduction and Herbrand-model-based simplifications [18, 24], and keeps STs close to the syntax and semantics of functional, logic and/or constraint languages. The informal models that guide the design of programs in such languages are in fact the "godfathers" of ST models.

The ST approach evolved from 25 years of research and development in the area of formal methods for software construction. It aims at keeping the balance between a wide range of applications and a simple mathematical foundation. To this end boundaries between many research communities had to be crossed. STs employ concepts, results and methods from many-sorted and modal logic, algebraic specification, term rewriting, automated theorem proving, structural operational semantics, functional and logic programming, fixpoint and category theory, universal algebra and coalgebra. Whatever was adopted from these areas, could be reformulated in terms of many-sorted logic with equality.

Formally, a swinging type starts out from *constructors* for visible sorts and *object constructors* for hidden sorts. Object constructors are, for instance, the injections into sorts denoting sums of hidden sorts. Each sort is equipped with a *structural* and a *behavioral equality*. For visible sorts, both equalities coincide. Constructors and object constructors build up data uniquely w.r.t. structural and behavioral equality, respectively. An ST defines functions, predicates and copredicates in terms of *generalized Horn clauses* and *co-Horn clauses*, respectively. A predicate is *static* or *dynamic*. Structural equalities are dynamic predicates with axiom expressing their congruence property. Behavioral equalities are copredicates with axioms expressing the compatibility with all *destructors* (which are defined functions or predicates) and the *zigzag compatibility* with all *transition predicates* (which are particular dynamic predicates). The latter implies that they are bisimulations.

The final model of a swinging type is a term model factored through its behavioral equalities. Unfortunately, such a model can only represent countably many elements even if the domain to be specified consists of uncountably many "infinite" objects like streams or processes. For representing such a domain an ST is extended by a *cospecification* that adds hidden sorts, destructors, copredicates, inductive or

[1] Appealing to modal-logic terminology, predicates and copredicates are also called μ- resp. ν-predicates.

coinductive axiomatizations of functions and *assertions* for defining a subdomain. The latter provide co-Horn axioms for membership predicates and thus complement the purpose of the co-Horn axioms for behavioral equalities that define a quotient domain. The cospecification has a final coalgebra that interprets a hidden sort as a set of *behaviors*. A behavior is given by a tuple of interpretations of *context* terms consisting of destructors. The construction of the final coalgebra generalizes the construction of the minimal automaton that realizes a fixed input-output behavior. Here the destructors state transition or output functions.

If a cospecification, say CSP, which already extends a swinging type, is augmented by further constructors, defined functions and predicates that involve the hidden sorts of CSP, the extension is called a *coalgebraic swinging type* (CST). Weak requirements ensure that the CST is a *conservative* extension of CSP, i.e. CST does not produce neither "junk" (data that are not already in the final coalgebra of CSP) nor "confusion" (theorems that do not already hold in that coalgebra). This property allows us to switch between algebraic (inductive) and coalgebraic, (coinductive) arguments when reasoning about CSTs.

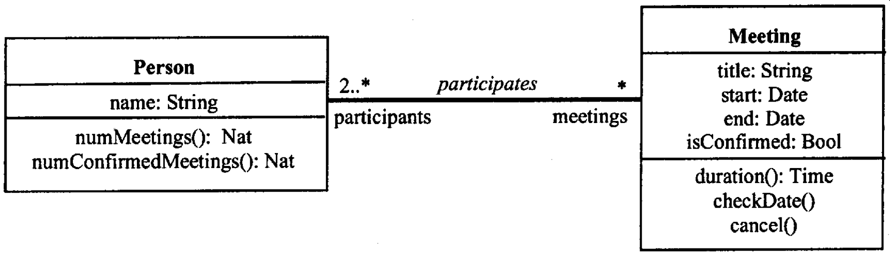


Figure 1. Two associated classes (Figure 3 of [11])

2 Class diagrams

[11] claims a general one-to-one correspondence between a class and a specification unit. However, a simple look at the graph structure of a class diagram reveals that this cannot work as soon as the graph involves cycles such as those created by bidirectional associations (cf. Fig. 1). A class diagram yields a specification, each class provides a (hidden) sort. The entire specification is built up hierarchically, following the use relationships between methods that form a, maybe collapsed, tree.

Even the finest specification structure reflecting a class diagram has to encapsulate all data and operations involved in a use cycle into a single specification unit. But this does not mean that the other extreme, recommended in [11], must be adopted, i.e., translating the entire class diagram into a single specification with a global state sort. Coalgebraic STs allow us to avoid the introduction of global state sorts that are not sums or products of sorts representing individual classes.

A UML class diagram can be translated into a CST as follows. A class becomes a hidden sort, say cl, of a cospecification. An **attribute** of cl with values in s yields a destructor $d : cl \rightarrow s$. A **method** $m(x_1 : s_1, \ldots, x_n : s_n)$ of cl is turned into a constructor $c : cl \times s_1 \times \ldots \times s_n \rightarrow cl$, while a method $m(x_1 : s_1, \ldots, x_n : s_n) : s$ of

cl provides a defined function $f : cl \times s_1 \times \ldots \times s_n \to s$. A **class-scope operation** $m(x_1 : s_1, \ldots, x_n : s_n)$ becomes a constructor $c : s_1 \times \ldots \times s_n \to cl$. Methods defined in terms of other methods may be introduced as defined functions.

An **association** $assoc$ that relates n classes $cl_1, \ldots, cl_n$ to each other can of course be regarded as an n-ary relation [7, 14, 28]. Then **rolenames** attached to the ends of $assoc$ correspond to attributes or projections in the sense of relational or algebraic models, respectively, and $assoc$ becomes a further hidden sort with membership $\in: (cl_1 \times \ldots \times cl_n) \times assoc$ as the only destructor. Binary and mostly **anonymous** associations, which provide the pathways for navigating between objects of the associated classes, however, should be translated differently. The relational view would enforce the computation of transitive closures of associations, which was shown to result in rather tricky and counter-intuitive code [14].

Hence a rolename r attached to the cl-end end of a binary association that relates cl to cl' is translated into a destructor d_r of the class cl' at the opposite end. If an association end lacks a rolename, we introduce one. The range sort s of d_r depends on the **multiplicity** at the cl-end, which holds r. If the multiplicity is 1, then $s = cl$. If the multiplicity is $m..n$, $+$ or $*$, then s is a sum sort: $\amalg_{i=m}^{n} cl^i$, $\amalg_{n>0} cl^n$ or $cl^* = \amalg_{n \in \mathbb{N}} cl^n$, respectively. Hence d assigns a list of cl-objects to each cl'-object. Additional constraints may demand another type of collection like a bag or a set. As long as proving that cl'-objects are behaviorally equivalent is not an issue, the actual collection type is irrelevant and we may keep to the lists given by the above sum sorts. Otherwise it is quite easy to turn these lists into bags or sets because finite (!) sets and bags also provide swinging types, though not coalgebraic ones.

In [27], binary associations are also translated into set-valued functions. But, since the authors do not give a semantics of objects or their states, it is not clear what the elements are the sets consist of. Translating a class into a constructor-based type whose objects are built up of attribute values and rolenames in a hierarchical way would be inadequate because associations may form cycles as in Fig. 1. Translating a class into a hidden sort of a CST, however, leads to a behavioral semantics in the sense described above: object states are interpretations of context expressions built up of destructors, here: of attributes and rolenames.

Example. The bidirectional association between the classes $Person$ and $Meeting$ in Fig. 1 suggests a single specification, but two sorts for person states and meeting states, respectively. The CST given below covers Fig. 1 as well as the following *OCL constraint* [30] taken from [11]:

```
context Meeting :: checkDate()
        post : isConfirmed =
               self.participants ->
               collect(meetings) ->
               forAll(m | m <> self and m.isConfirmed implies
                         (after(self.end,m.start) or (after(m.end,self.start)))
```

For dealing with object (state) sets we start out from a parameterized specification FINSET of finite subsets of a set of instances of a class cl (cf. [23]). FINSET is an ST with the only set-destructor $in : cl \times set(cl) \to bool$ that denotes set membership. FINSET includes a defined function $mkset : cl^* \to set(cl)$ that transforms sequences into sets.

PERSON&MEETING = FINSET `and` STRING `and` DATE&TIME `then`

hidsorts	$Person\ \ Meeting$
destructs	$name : Person \to String$
	$meetings : Person \to Meeting^*$
	$title : Meeting \to String$
	$participants : Meeting \to Person^*$
	$start, end : Meeting \to Date$
	$isConfirmed : Meeting \to bool$
	$\pi_i^n : Person^n \to Person$
	$\pi_i^n : Meeting^n \to Meeting$ for all $1 \leq i \leq n \in \mathbb{N}$
constructs	$checkDate : Meeting \to Meeting$
	$cancel : Meeting \to Meeting$
defuncts	$Meetings : Person \to set(Meeting)$
	$Participants : Meeting \to set(Person)$
	$numMeetings : Person \to nat$
	$numConfirmedMeetings : Person \to nat$
	$duration : Meeting \to Time$
	$consistent : Meeting \times Meeting \to bool$
vars	$p : Person\ \ m, m' : Meeting\ \ ms : set(Meeting)\ \ ps : set(Person)$
Horn axioms	$Meetings(p) \equiv mkset(meetings(p))$
	$Participants(m) \equiv mkset(participants(m))$
	$numMeetings(p) \equiv \vert Meetings(p)\vert$
	$numConfirmedMeetings(p) \equiv \vert filter(isConfirmed, Meetings(p))\vert$
	$duration(m) \equiv end(m) - start(m)$
	$consistent(m, m') \equiv not(isConfirmed(m'))$
	$\qquad or\ end(m) < start(m')\ or\ end(m') < start(m)$
	$isConfirmed(checkDate(m)) \equiv forall(\lambda m'.consistent(m, m'), remove(m, ms))$
	$\quad \Leftarrow\ Participants(m) \equiv ps \wedge flatten(map(Meetings, ps)) \equiv ms$
	$isConfirmed(cancel(m)) \equiv false$
	$\pi_i^n(p_1, \ldots, p_n) \equiv p_i$
	$\pi_i^n(m_1, \ldots, m_n) \equiv m_i$ for all $1 \leq i \leq n \in \mathbb{N}$
assertions	$\vert Participants(m)\vert \geq 2$

Basic methods are declared as constructors, whereas derived ones are declared as defined functions. An element of the final coalgebra of PERSON&MEETING (informally: a state or behavior of a PERSON&MEETING-object) is the tuple of values of all PERSON&MEETING-contexts, i.e. non-hidden-sorted terms over the above destructors. For instance, a meeting with five participants whose third one is called Henry is represented by a tuple of *meeting*-context values such that the context $name(\pi_3^5(participants(m)))$ evaluates to Henry. ❏

More details about the translation of UML classes into CSTs, in particular the treatment of generalizations (inheritance), can be found in [23] and [20]. The integration of state machines in terms of transition predicates and the proof of state reachability or invariance are also topics of [23].

[17] and [18] present the model- and proof-theoretical foundations of non-coalgebraic STs. [19] and [21] focus on their structured development (including refinements) and the emdedding of cospecifications, respectively.[2] For introductions to hidden algebras, coalgebras and coinduction, we recommend [13,25,29] and [10]. For the specification of coalgebras or transitions systems, see also [6], [5] and [12]. [23] explores various application areas for STs and attempts to integrate certain traditional formal

[2] These two papers are under revision and will probably re-appear under slightly different titles.

description and proof techniques into the ST framework. Examples can also be found in my course notes [22]. [24] provides a survey of proof and computation rules for reasoning about STs. Our proof and symbolic-computation system *Expander2* [16] has been designed in particular for the purpose of carrying out and visualizing such reasoning semi-automatically.

References

1. E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Report, Springer 1999
2. E. Astesiano, G. Reggio, *Algebraic Specification of Concurrency*, Proc. WADT'91, Springer LNCS 655 (1993) 1-39
3. M. Bidoit, R. Hennicker, M. Wirsing, *Behavioural and Abstractor Specifications*, Science of Computer Programming 25 (1995) 149-186
4. J.C.M. Baeten, W.P. Weijland, *Process Algebra*, Cambridge University Press 1990
5. C. Cîrstea, *A Coequational Approach to Specifying Behaviours*, Proc. CMCS '99, Elsevier ENTCS 19 (1999)
6. G. Costa, G. Reggio, *Specification of Abstract Dynamic Data Types: A Temporal Logic Approach*, Theoretical Computer Science 173 (1997) 513-554
7. B. Demuth, H. Hußmann, *Using UML/OCL Constraints for Relational Database Design*, Proc. UML '99, 1999
8. J.A. Goguen, R. Diaconescu, *An Oxford Survey of Order Sorted Algebra*, Mathematical Structures in Computer Science 4 (1994) 363-392
9. J.A. Goguen, G. Malcolm, *Hidden Coinduction*, Mathematical Structures in Computer Science 9 (1999) 287-319
10. J.A. Goguen, G. Malcolm, *A Hidden Agenda*, UCSD Technical Report CS97-538, San Diego 1997, www-cse.ucsd.edu/users/goguen/ps/ha.ps.gz
11. H. Hußmann, M. Cerioli, G. Reggio, F. Tort, *Abstract Data Types and UML Models*, Report DISI-TR-99-15, University of Genova 1999
12. B. Jacobs, *Exercises in Coalgebraic Specification*, Report, Dept. Comp. Sci., University of Nijmegen 1999
13. B. Jacobs, J. Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, EATCS Bulletin 62 (1997) 222-259
14. M. Mandel, M.V. Cengarle, *On the Expressive Power of the Object Constraint Language OCL*, Proc. FM '99, Springer LNCS 1708 (1999) 854-874
15. J. Meseguer, *Membership Algebra as a Logical Framework for Equational Specification*, Proc. WADT '97, Springer LNCS 1376 (1998) 18-61
16. P. Padawitz, *Expander2: A System for Designing and Verifying Functional-Logic Programs and Visualizing Proofs and Computations*, ls5.cs.uni-dortmund.de/~peter/Expander2.html, University of Dortmund 2001
17. P. Padawitz, *Proof in Flat Specifications*, in [1]
18. P. Padawitz, *Swinging Types = Functions + Relations + Transition Systems*, Theoretical Computer Science 243 (2000) 93-165
19. P. Padawitz, *Modular Swinging Types*, Report, University of Dortmund 1999, ls5.cs.uni-dortmund.de/~peter/MST.ps.gz
20. P. Padawitz, *Swinging UML: How to Make Class Diagrams and State Machines Amenable to Constraint Solving and Proving*, Proc. UML 2000, Springer LNCS 1939 (2000) 162-177
21. P. Padawitz, *Swinging Types II: Hierarchical and Coalgebraic Specifications*, Report, University of Dortmund 2000, ls5.cs.uni-dortmund.de/~peter/BehCoalg.ps.gz
22. P. Padawitz, *Theorie der Programmierung*, Course Notes, University of Dortmund 1998-2000,
ls5.cs.uni-dortmund.de/~peter/TdP96.ps.gz

23. P. Padawitz, *Sample Swinging Types*, Report, University of Dortmund 2000, ls5.cs.uni-dortmund.de/∼peter/BehExa.ps.gz
24. P. Padawitz, *Basic Inference Rules for Algebraic and Coalgebraic Specifications*, Report, University of Dortmund 2001, ls5.cs.uni-dortmund.de/∼peter/WADT01.ps.gz
25. H. Reichel, *An Approach to Object Semantics based on Terminal Coalgebras*, Math. Structures in Comp. Sci. 5 (1995) 129-152
26. H. Reichel, *Unifying ADT- and Evolving Algebra Specifications*, EATCS Bulletin 59 (1996) 112-126
27. M. Richters, M. Gogolla, *On Formalizing the UML Object Constraint Language OCL*, in: Proc. Conceptual Modeling - ER '98, Springer LNCS 1507 (1998) 449-464
28. G. Roşu, J.A. Goguen, *Circular Coinduction*, UCSD Report, San Diego 2000, www-cse.ucsd.edu/users/goguen/ps/ccoind.ps.gz
29. J.J.M.M. Rutten, *Universal Coalgebra: A Theory of Systems*, Report CS-R9652, CWI, SMC Amsterdam 1996
30. J.B. Warmer, A.G. Kleppe, *The Object Constraint Language*, Addison-Wesley 1999

Embedding Processes in a Declarative Programming Language*

Bernd Braßel[1], Michael Hanus[2], and Frank Steiner[2]

[1] RWTH Aachen, Germany
brassel@halifax.rwth-aachen.de
[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh,fst}@informatik.uni-kiel.de

Abstract While declarative programming languages are based on the idea of specifying the static relationships of problems, the right modeling of the dynamic behavior is equally important for many practical applications. To support a high-level specification of both aspects of computational systems, we propose the embedding of a process-oriented specification language in a multi-paradigm declarative language. Since this embedding is done in a seamless way, the features of the declarative base language can be exploited (1) for a high-level specification of the computational needs in single state transitions of a dynamic system, and (2) to reuse the abstraction facilities of the base language for the specification of the structure of dynamic systems. We show an implementation of these ideas in the declarative multi-paradigm language Curry. This implementation has been used for a prototypical implementation of embedded and distributed systems in a high-level manner.

1 Introduction

Declarative programming languages (e.g., functional, logic, or functional logic languages) aim to support *high-level* descriptions of software systems, which are *executable* at the same time. Such programming languages have many advantages w.r.t. the efficient development, reliability, maintenance, analysis, and verification of programs. The general idea of declarative programming is the specification of the static relationships of a given problem by well-understood mathematical entities (functions and/or predicates). However, many real-world applications demand also for an appropriate modeling of the dynamic behavior of a complex software system, which may be distributed into communicating active parts or embedded in an environment where they must react on external events.

Processes are an appropriate notion to describe dynamic behavior, and high-level formalisms [3] have been developed to describe the essence of process behaviors, like communication, parallelism, process creation, on an abstract level. We want to keep the advantages of declarative programming but make them applicable also for a wider range of applications, like distributed or embedded systems. For this purpose, we propose an embedding of a process-oriented specification language in a declarative language, where we choose the declarative multi-paradigm language Curry [4, 9] as our base. On the one hand, our proposal is based on a clear separation between the declarative and dynamic aspects of an application system by making processes a distinguished data type (this has some similarities to the separation of pure functions and functions manipulating the external world by the introduction of monads [12]).

* This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2 and by the DAAD under the PROCOPE programme.

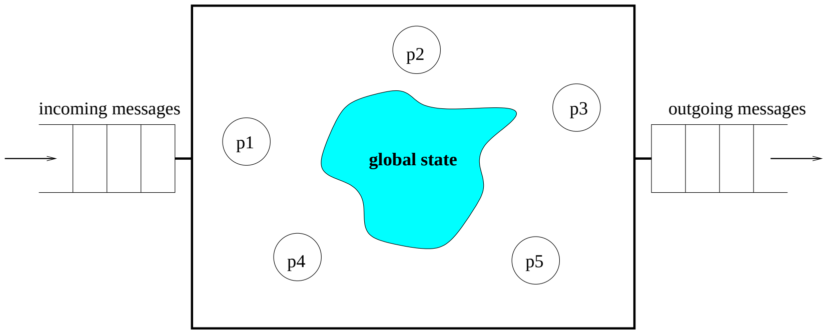


Figure 1. Component of a dynamic system

On the other hand, our embedding of processes is done in a seamless way by defining processes as a standard data type so that the features of the base language can be exploited in two ways: (1) the computational needs in single state transitions of a dynamic system can be specified in a high-level style, and (2) the abstraction facilities of the base language can be reused for the specification of the structure of dynamic systems.

Our objective is to provide a domain-specific language for the description of dynamic systems that should react on external events (e.g., embedded systems). The integration of such a domain-specific language into an existing high-level programming language has the advantage that we can reuse the functionality of the base language in application programs and we can provide a prototypical implementation of the entire framework with a limited effort. We have used our implementation for a prototypical implementation of embedded and distributed systems in a high-level manner.

This paper is structured as follows. In the next section we provide a basic introduction into our framework. Section 3 sketches the features of Curry as necessary for the understanding of this paper. In Section 4 we show the modeling of our process-oriented specifications in Curry. Its application is demonstrated in Section 5 by several examples before we make some remarks about the implementation of our framework in Section 6 and conclude in Section 7.

2 Specification of Process Systems

Our framework is based on the partition of a dynamic system into several components that cooperate by exchanging messages. In an embedded system, such a component could correspond to a controller that reacts on messages received from external sensors by sending messages to other active components. In a distributed software system, these components may run on different computers and exchange messages via the Internet. Our goal is to provide a framework for the high-level description of such components.

The structure of each component is sketched in Fig. 1. A *component* consists of a set of processes (p1,p2,...), a global state (i.e., data visible for all processes inside a component but not visible from outside) and a mailbox (queue of messages sent to this component). The behavior of a dynamic system is defined by the behavior of each process. A process can be activated depending on conditions on the global state or the mailbox. If a process is activated (e.g., because a particular message arrives in the mailbox), it performs actions and may start other processes (since we have an interleaving semantics, at most one process can perform actions so that actions are atomic entities).

The reaction of a process to the change of its external context (i.e., mailbox or global state) consists of a sequence of *actions*. Currently, possible actions are the change of the global state, the sending of a message to another component,[1] or the removing of a message from the mailbox. (Note that messages are not automatically removed after reading since there may be several processes that must react on the same message.) Of course, the set of possible actions can be extended but our current set is sufficient for our case studies.

The *global state* of a component can be accessed and manipulated by all processes of the same component. Thus, it also serves as a facility for process synchronization. In general, the global state is just a tuple of data items. However, our case studies have shown that it is quite useful to partition the state into a static part with a fixed number of items and a dynamic part with an evolving number of items.[2] Therefore, we provide different actions to manipulate the static or the dynamic part of the state. The static part is changed by defining a new value for it (where changes to single items could be expressed by record updates) ("`SetState` s"). For the manipulation of the dynamic part, there are two actions: one for creating a new item ("`NewName` v *ref*"), which can subsequently be accessed via the newly created name *ref*, and one for changing the value associated to a dynamic item ("*ref* `:=` v"). Furthermore, there is a function `get` to extract the associated value of a dynamic item in a store. Thus, the following table summarizes the current set of actions:

`Send` m	send message m
`SetState` s	set static state to s
ref `:=` v	set dynamic state object *ref* to value v
`NewName` v *ref*	create new dynamic state object *ref* with initial value v
`Deq` m	remove message m from mailbox

As described above, *processes* are activated, depending on a particular condition on the mailbox and/or global state, and perform an action followed by the creation of new processes. Thus, the behavior of each process is specified by

- a *guard* (i.e., a condition on the mailbox and/or state),
- a sequence of *actions* (to be performed when the guard is satisfied and the process is selected for execution), and
- a *process term* describing the further activities after executing the actions.

[1] For the sake of simplicity, all outgoing messages are sent via the same channel. If a component wants to send messages to different other components, the messages must be tagged (to identify the receiving component) and it is the purpose of a distributor connecting the different components to forward the outgoing messages to the right receiver. Of course, one can extend our model so that a component can send messages directly to different other components but we have made the experience that our restricted model provides more modularity.

[2] Note that this distinction is not strictly necessary, since the static part can also contain dynamic data structures like lists, but useful to structure components (see the multiple counter example in Section 5).

In order to structure dynamic system specifications in an appropriate manner, we allow *parameterized processes* since this supports the distinction between *local and global state*: process parameters are only accessible inside a process and, therefore, they correspond to the local state of a process, whereas the global state is visible to all processes inside a component. Changes to the local state can simply be obtained by recursive process calls with new arguments. Thus, the language of *process terms* p is very similar to process algebra [3] and defined by the following grammar:

$$
\begin{array}{rll}
p ::= & \texttt{Terminate} & \text{successful termination} \\
\mid & \texttt{[}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{]} & \text{sequence of actions} \\
\mid & \texttt{p}\ t_1 \ldots t_n & \text{run process } \texttt{p} \text{ with parameters } t_1 \ldots t_n \\
\mid & p_1\ \texttt{>>>}\ p_2 & \text{sequential composition} \\
\mid & p_1\ \texttt{<|>}\ p_2 & \text{parallel composition} \\
\mid & p_1\ \texttt{<+>}\ p_2 & \text{nondeterministic choice} \\
\mid & p_1\ \texttt{<\%>}\ p_2 & \text{nondeterministic choice with priority} \\
\mid & p_1\ \texttt{<\~{}>}\ p_2 & \text{parallel composition with priority}
\end{array}
$$

A sequence of actions is executed from left to right as one atomic operation (having a sequence of actions instead of one single action is useful to specify larger critical regions in many applications, e.g., see the dining philosophers example below). The operators "`>>>`", "`<|>`", and "`<+>`" are standard in process algebra, whereas the last two operators are not very common but useful in applications where a simple nondeterministic choice is not appropriate. The meaning of "p_1 `<%>` p_2" is: "If process p_1 can be executed, execute p_1 (and remove p_2), otherwise execute process p_2 (and remove p_1), if possible." The meaning of "p_1 `<~>` p_2" is: "Execute processes p_1 and p_2 in parallel (like "p_1 `<|>` p_2") but p_2 is executed only if p_1 cannot be executed; if p_1 terminates, then also p_2 terminates." The latter combinator is useful for idle background processes like concurrent garbage collectors.

Using this language of process terms, the behavior of a parameterized process is defined by a *process abstraction* of the following form:

$$
\begin{array}{ll}
\texttt{p}\ x_1 \ldots x_n & \mid\ guard_1\ \texttt{=}\ actions_1\ \texttt{>>>}\ p_1 \\
 & \ \vdots \\
 & \mid\ guard_k\ \texttt{=}\ actions_k\ \texttt{>>>}\ p_k
\end{array}
$$

where $guard_i$ is a decidable condition on the mailbox, the global state and the process parameters, $actions_i$ is a sequence of actions, and p_i is a process term ($i = 1, \ldots, k$). The different guards together with their right-hand sides are considered to be combined with the "`<%>`" operator, i.e., the first alternative with a valid guard is selected for execution.

As a first example for the use of our framework, consider the classical "dining philosophers". The global state in this example has only a static component, namely the list (or array) of `forks` where each fork has either the value `Avail` ("available") or `Used`. The entire component consists of processes `Thinking` or `Eating` that are parameterized by the number of the philosopher. Then the complete specification is as follows (`forks[`i`]` denotes the value of the i-th fork and `forks[`i`<-`v`]` denotes a new state identical to `forks` but with the value v for the i-th component):

```
Thinking i | forks[i]==Avail ∧ forks[i+1 mod n]==Avail
           = [SetState forks[i<-Used, i+1 mod n <- Used]]
             >>> Eating i
```

```
Eating i    = [SetState forks[i<-Avail, i+1 mod n <- Avail]]
              >>> Thinking i
```

Thus, if philosopher i is thinking (which corresponds to the existence of a process term "`Thinking` i") and both forks are available, then he can use both forks to turn into the `Eating` process. Note that the change of the global state, i.e., the use of both forks, can only be performed (in an atomic manner) if both forks are really available. Therefore, the classical deadlock situation is avoided without low-level synchronization (e.g., semaphores) or additional constructions (e.g., room tickets).

An example where a global state with a dynamic part becomes important will be shown later. Next we will show how this specification language can be embedded into the declarative multi-paradigm language Curry in order to obtain an executable specification language for modeling dynamic systems. Before doing so, we review the basic elements of Curry.

3 Curry

In this section we survey the elements of Curry which are necessary to understand the design and implementation of our language for specifying processes. More details about Curry's computation model and a complete description of all language features can be found in [4,9].

Curry is a modern multi-paradigm declarative language combining in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, and encapsulated search). From a syntactic point of view, a Curry program is a functional program[3] extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Thus, a Curry program consists of the definition of functions and the data types on which the functions operate. Functions are evaluated in a lazy manner. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. The behavior of function calls with free variables depends on the evaluation annotations of functions which can be either *flexible* or *rigid*. Calls to rigid functions are suspended if a demanded argument, i.e., an argument whose value is necessary to decide the applicability of a rule, is uninstantiated ("*residuation*"). Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments to the required values in order to apply a rule ("*narrowing*").

Example 1. The following Curry program defines the data types of Boolean values and polymorphic lists (first two lines) and a function to compute the concatenation of two lists:

```
data Bool   = True | False
data List a = []   | a : List a

conc :: [a] -> [a] -> [a]
conc eval flex
```

[3] Curry has a Haskell-like syntax [10], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

```
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys
```

The data type declarations introduce `True` and `False` as constants of type `Bool` and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell).

The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[4] Since `conc` is explicitly defined as flexible[5] (by "`eval flex`"), an equation "`conc ys [x] =:= xs`" can be solved by instantiating the first argument `ys` to the list `xs` without the last argument, i.e., for a given `xs`, the only solution to this equation satisfies that `x` is the last element of `xs`.

In general, functions are defined by (*conditional*) *rules* of the form "l | c = e" where l has the form $f\, t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, the *condition* c (which can be omitted) is a constraint, and e is a well-formed *expression* which may also contain function calls, lambda abstractions etc. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable. A *constraint* is any expression of the built-in type `Success`. Each Curry system provides at least equational constraints of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms (i.e., terms without defined function symbols). In contrast, e_1 `==` e_2 denotes an *equality test* which is successful only if both sides e_1 and e_2 are reducible to identical *ground* data terms, i.e., the test suspends in the presence of free variables.

The operational semantics of Curry, precisely described in [4, 9], is based on an optimal evaluation strategy [1] and can be considered as a conservative extension of lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Concurrent programming is supported by a concurrent conjunction operator "`&`" on constraints, i.e., a non-primitive constraint of the form "c_1 `&` c_2" is evaluated by solving both constraints c_1 and c_2 concurrently. Furthermore, distributed programming is supported by ports [5] which allows the sending of arbitrary data terms (also including logic variables) between different computation units possibly running on different machines connected via the Internet. The port concept has been used to integrate object-oriented features into Curry [8] and for high-level GUI (Graphical User Interface) programming in Curry [6]. Furthermore, it is relevant for the work described in this paper since the different components of a dynamic system communicate via ports (which is, however, not directly visible to the programmer).

4 Specification of Process Systems in Curry

Now we are ready to define an implementation of process-oriented specifications, as introduced in Section 2, in Curry. The implementation is guided by the motivation to enable the writing of specifications in the high-level style of Section 2. The main

[4] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

[5] As a default, all functions except for constraints are rigid.

difference (and advantage!) is the fact that Curry is a typed language (so that we have a type checker for specifications for free) and allows definitions by pattern matching.

First, we introduce the languages of actions and process terms as data types in Curry. The following data type declaration defines the possible actions. `ObjRef` is an abstract data type denoting references to dynamic objects, and *inmsg*, *outmsg*, *static*, and *dyn* are type variables denoting the type of incoming messages, outgoing messages, the static part and the dynamic items of the global state in a concrete specifcation.

```
data Action inmsg outmsg static dyn =
          Send outmsg          -- send message
        | SetState static      -- set static state
        | Assign ObjRef dyn    -- set dynamic state object
        | NewName dyn ObjRef   -- create new dynamic object
        | Deq inmsg            -- remove message from mailbox
```

In order to support the same notation as in Section 2, we define the following function (infix operator) as a synonym for the `Assign` action:

```
ref := cont  =  Assign ref cont
```

The data type of process terms has a similar definition but with the type *proc* of concrete processes as an additional type parameter:

```
data ProcExp proc inmsg outmsg static dyn =
                 Terminate
               | Atomic [Action inmsg outmsg static dyn]
               | Proc proc
               | ParProc   (ProcExp proc inmsg outmsg static dyn)
                           (ProcExp proc inmsg outmsg static dyn)
               | SeqProc   (ProcExp proc inmsg outmsg static dyn)
                           (ProcExp proc inmsg outmsg static dyn)
               | ChProc    (ProcExp proc inmsg outmsg static dyn)
                           (ProcExp proc inmsg outmsg static dyn)
               | ChPriProc (ProcExp proc inmsg outmsg static dyn)
                           (ProcExp proc inmsg outmsg static dyn)
               | ParIdle   (ProcExp proc inmsg outmsg static dyn)
                           (ProcExp proc inmsg outmsg static dyn)
```

Again, we support the same notation as in Section 2 by the following operator definitions:

```
p1 >>> p2  =  SeqProc   p1 p2
p1 <|> p2  =  ParProc   p1 p2
p1 <+> p2  =  ChProc    p1 p2
p1 <%> p2  =  ChPriProc p1 p2
p1 <~> p2  =  ParIdle   p1 p2
```

In order to exploit the language features of Curry for the specification of dynamic systems, we consider a *system specification* as a mapping which assigns to each process, mailbox (list of incoming messages), static and dynamic state (list of dynamic objects) a process term (similarly to Haskell, a `type` definition introduces a type synonym in Curry):

```
type Specification proc inmsg outmsg static dyn =
              proc -> [inmsg] -> static -> [DynObj dyn]
                   -> ProcExp proc inmsg outmsg static dyn
```

This definition has the advantage that one can use standard function definitions by pattern matching for the specification of systems, i.e., one can define the behavior of processes in the following form:[6]

```
spec (p x1...xn) mailbox state refs
                | < condition on x1,...,xn, mailbox, state, refs >
                = Atomic [actions] >>> process term
```

Hence, the guard is just a standard constraint on the parameters $x_1, \ldots, x_n$, `mailbox`, `state`, and `refs` so that we need no global variables or auxiliary constructs to access the current global state or mailbox (note that the access to these entities was left unspecified in Section 2).

As an example we show the complete specification of the dining philosophers of Section 2. It consists of the definition of data types for the values of forks, the philosopher processes and the definition of the specification function `phil_spec` ("`rpl` l i v" replaces the i-th element of the list l by v):

```
data ForkStatus = Avail | Used

data PhiloProc = Eating Int | Thinking Int

n = 5   -- here we have five philosophers

phil_spec (Thinking i) _ forks _
 | forks!!i == Avail && forks!!((i+1)`mod`n) == Avail
 = Atomic [SetState (rpl (rpl forks i Used) ((i+1)`mod`n) Used)]
   >>> Proc (Eating i)

phil_spec (Eating i) _ forks _ =
 Atomic [SetState (rpl (rpl forks i Avail) ((i+1)`mod`n) Avail)]
 >>> Proc (Thinking i)
```

Note that neither the mailbox nor the dynamic part of the state is used in this simple example. Initially, all philosophers are thinking. This can be expressed by a process term where five philosopher processes are combined in parallel:

```
phils = foldr1 (<|>) (map (\i->Proc (Thinking i)) [0..n-1])
```

Note that we can use standard higher-order functions like `foldr1` or `map` to create complex process terms since process terms are first-order objects in our specification language. Hence, the expression `phils` reduces to the term

```
Proc (Thinking 0) <|> ··· <|> Proc (Thinking 4)
```

A system specification is executed by providing

1. a port name for incoming messages
2. a port name for outgoing messages
3. an initial process term
4. a system specification
5. an initial (static) state[7]

[6] Since we allow arbitrary Boolean expressions or constraints as conditions, the decidability of the guard is not automatically given but must be ensured by the programmer.

[7] The dynamic part of the state is always empty at the beginning.

This is the purpose of the main function `exec_system` so that we can execute our specification as follows:

```
exec_system "in"  "out"  phils  phil_spec
            (take n (repeat Avail))  -- all forks are available
```

The main advantage of our embedding of a process-oriented language in Curry (rather than defining a complete new specification language) is the reuse of the features of Curry for the specification language, in particular:

- The type checker of Curry can also be used to type check specifications and detect inconsistencies in specifications.
- Functional programming is useful to compute values in actions, process parameters, new states etc.
- Constraint programming is useful for checking complex conditions.
- The standard abstraction facilities of Curry (e.g., higher-order functions) are useful to structure the specification of dynamic systems, in particular, we can define functions to compute process terms (compare `phils` above).

5 Examples

Due to lack of space, we can only sketch two further examples that are implemented using our framework. The first example is a challenge from the Glasgow Research Festival[8], a system of multiple counters.

The application starts be creating a single window, as shown to the right, that visualizes a counter. This counter can be manually (button "Inc") or automatically (periodically) incremented (after pressing the button "Auto"). Pressing the "Copy" button creates a new counter with its own independent state, and pressing the "Link" button creates a new view (counter window) to the same counter.

We implement this system by the specification of a *counter control system* which is responsible to control all counters and organize the communications with the different windows. If the user presses a button in a window *win*, an appropriate message (e.g., `(Inc` *win*`)`, `(Copy` *win*`)`) is sent to the counter controller which must correctly react to this request. The global state of the counter controller has only a dynamic part since a new counter object is created in the state whenever the users presses the "Copy" button. The value of a counter object has the form `(Counter` *val wins mode*`)` where *val* is the current value of the counter, *wins* is a list of windows where this counter is displayed, and *mode* is the increment mode of the counter (`Manual` or `Automatic`). As a consequence, the individual processes of the controller are parameterized with references to counter objects and windows. For instance, there is a process `(Manual_Ctrl c w)` for each counter object `c` and window `w` where `c` is displayed. This process is responsible for processing the messages received from window `w`. Using pattern matching, there is one rule for each message in the specification. For instance, the rule for the message `Inc` is as follows:

```
cctrl (Manual_Ctrl c w) (Inc win:_) _ store | win==w
 = let Counter val windows _ = get c store in
   Atomic [c := Counter (val+1) windows Manual, Deq (Inc win)]
```

[8] `http://www.cs.chalmers.se/˜magnus/GuiFest-95/`

```
    >>> Proc (Refresh_Window c) <|> Proc (Manual_Ctrl c w)
```

Thus, the guard consists of checking whether the message comes from the window for which this process is responsible. If this is the case, the value of the counter is incremented (where the increment mode is set to `Manual`), the message is removed from the mailbox, and a new process for refreshing all windows for this counter is created. The latter process sends update messages to all appropriate windows, where we apply some standard higher-order functions:

```
cctrl (Refresh_Window c) _ _ store =
   let Counter val windows _ = get c store in
   foldr (<|>) Terminate
         (map (\w->Atomic [Send (Update w val)]) windows)
```

The remaining cases are similarly defined. In particular, for each counter object `c` there is a process `(Automatic_Ctrl b c)` which is responsible for incrementing counters in automatic mode. This is done by an external clock which sends clock ticks as messages to the controller so that the `Automatic_Ctrl` processes are activated on these messages. Since there may be many of these processes, the clock tick message should not be deleted in the mailbox by any of these processes (since all of them must have the chance to react). This is the purpose of a background process `Delete_Clocks` which is simply defined as (each clock signal has a Boolean flag to distinguish successive signals):

```
cctrl Delete_Clocks (ClockSignal flag:_) _ _ =
   Atomic [Deq (ClockSignal flag)] >>> Proc Delete_Clocks
```

The complete specification of the counter controller, which is omitted due to lack of space, consists of nine rules (in addition to the three rules above, four further rules for handling the counter button messages, one rule for the `Automatic_Ctrl` process and one rule for the `Create_Window` process that creates a window together with a `Manual_Ctrl` process for it) which specify in a high-level and readable way the behavior of all processes in the controller. The initial configuration is defined by the following process term (where `c` is a free variable denoting the reference to the first counter object created by `NewName`):

```
Atomic [NewName (Counter 0 [] Manual) c] >>>
((Proc (Create_Window c) <|> Proc (Automatic_Ctrl True c))
 <~> Proc Delete_Clocks)
```

Note that it is important to create the process `Delete_Clocks` as a background process with lowest priority so that the clock signals are deleted only if no other process can be active. The remaining parts of the complete implementation, namely the counter GUIs, are also only a few lines of code thanks to the use of the Curry library for high-level GUI programming [6].

Our second example is a lift control system as visualized in Fig. 2. It consists of a number of request buttons that are inside a lift (left) or outside on the different floors (right), and a lift that can move up and down as well as open and close the doors. Instead of controlling a real lift, we simulate the lift also as a dynamic system. Thus, our implementation consists of two components in the sense of Fig. 1: a *lift controller* that accepts requests from the buttons, reacts on sensor messages from the lift (e.g., arrival at some floor), and sends appropriate control commands to the lift unit, and a *lift simulator* which simulates the lift by reacting on commands from

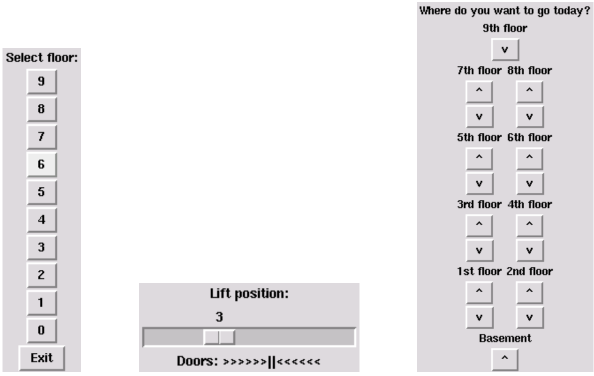


Figure 2. A lift control system

the lift controller and sending sensor messages to the controller and the GUI (shown in the middle of Fig. 2).

The specification of the entire system can be appropriately expressed in our framework. For instance, the lift controller consists of two processes running in parallel: a process `Sorting` which is responsible to react on user requests by computing a list of floors where the lift should stop (this list is sorted according to the movement of the lift), and a second process for controlling the lift. This process can be either `Moving` or `Stopped` according to the state of the lift unit (e.g., `Moving` waits for sensor messages from the lift unit about the reached floor, and `Stopped` waits for floor requests put in by `Sorting` in the global state).

Due to lack of space, we cannot show further details from this specification, but the complete implementation is available from the authors.

6 Implementation

Our process-oriented specification language is implemented as a standard Curry library so that it can be used in any Curry program. It is freely available as a library for PAKCS (Portland Aachen Kiel Curry System) [7] and completely implemented in Curry, using the features for distributed programming [5] to implement the communication between different components of a system. The current implementation is based on an interpreter for process terms according to the operational semantics of dynamic systems (see also [2]). Although the interpreter approach is not very efficient, it is fast enough to run our examples and required only a limited implementation effort (the complete implementation consists of approximately 200 lines of Curry code, without the imported standard libraries of Curry).

7 Conclusions

We have presented a domain-specific language for process-oriented programming. Since this language is embedded in the declarative multi-paradigm language Curry, we enable process-oriented programming in Curry, which is useful for the implementation of distributed or embedded systems. On the other hand, we can reuse the programming language features of Curry for the high-level specification of dynamic systems. The specification language is based on process algebras and offers parameterized processes and a global store for the exchange of data between processes. Thus, all internal communication (synchronization) between processes is performed via the store, whereas the external communication between different dynamic systems is done by sending messages. Although our language allows high-level specifications as in other process-oriented specification languages, it is *fully executable* at the same time. Therefore, it is a useful tool to implement and test dynamic systems in a prototypical manner. We have shown the appropriateness of our framework by several case studies.

There are many proposals for process-oriented specification languages (for example, see [3]). However, as far as we know, our work is the first fully implemented approach to exploit the high-level features of both functional and logic programming for process-oriented specifications. The most similar proposal to our approach is [2] (which is not accidental since our work is inspired by many discussions with the authors of [2]). Therefore, we refer to [2] for a detailed discussion of related work. [2] contains a "generic" framework for the extension of declarative (functional, logic, functional logic) languages to include processes where there is a strict distinction between the language of processes and the underlying programming language. In particular, declarative programs are considered as the global state between transition steps of processes. Thus, the modification of declarative programs are allowed without restrictions, i.e., arbitrary program clauses can be added or deleted. This complicates the implementation of their framework. Moreover, when modifying values associated to names, the evaluation time becomes important, but this is not clearly specified in their framework. To provide an effective implementation, we have restricted all modifications to a set of well-defined data items (partitioned into a static and dynamic part of the global store). As shown by our case studies, this is sufficient for all examples discussed in [2]. Moreover, we could provide fully executable specifications of all examples, which is due to the use of the Curry libraries for distributed [5] and GUI [6] programming.

For future work we will consider more applications to study the appropriateness of our approach or necessary extensions (like real-time conditions). Furthermore, it would be interesting to consider the translation of our specification language into other existing specification or control languages in order to reuse existing verification or implementation frameworks. This would enable the use of high-level declarative programming techniques in new application fields.

Acknowledgements. The authors are grateful to Rachid Echahed and Wendelin Serwe for fruitful discussions that led to the development described in this paper.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
2. R. Echahed and W. Serwe. Combining Mobile Processes and Declarative Programming. In *Proc. of the 1st International Conference on Computation Logic (CL 2000)*, pp. 300–314. Springer LNAI 1861, 2000.
3. W. Fokkink. *Introduction to Process Algebra.* Springer, 2000.
4. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.

5. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 376–395. Springer LNCS 1702, 1999.
6. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
7. M. Hanus, S. Antoy, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2000.
8. M. Hanus, F. Huch, and P. Niederau. An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pp. 89–106. Springer LNCS 2011, 2001.
9. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
10. J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.
11. A. Podelski and G. Smolka. Operational Semantics of Constraint Logic Programs with Coroutining. In *Proc. of the Twelfth International Conference on Logic Programming (ICLP'95)*, pp. 449–463. MIT Press, 1995.
12. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

Implementierung von *Port-basiertem Distributed Haskell*

Volker Stolz[1] und Frank Huch[2]

[1] RWTH Aachen, 52056 Aachen
[2] Christian-Albrechts-Universität zu Kiel, 24118 Kiel

Zusammenfassung In diesem Artikel wird die Implementierung eines robusten verteilten und offenen Systems in Haskell unter Zuhilfenahme von Ports vorgestellt. Dieses System erlaubt einer beliebigen Anzahl von Anwendungen miteinander zu kommunizieren. Eine Anwendung kann ihre Dienste beliebigen Clients im Netzwerk anbieten, so daß diese über das Internet darauf zugreifen können. Das Laufzeitsystem bietet einen Namensdienst, welcher symbolische Namen auf die intern verwendeten Kommunikationsstrukturen abbildet. Mittels einer Stream-Schicht wird von dem eigentlichen Lesen der Nachrichten abstrahiert. Sie erlaubt es, einen Teil der Kommunikation in einem funktionalen Kontext abzuwickeln.

1 Einführung

Heutzutage erfreuen sich verteilte Systeme in Form der sogenannten *peer-to-peer*-Anwendungen immer größerer Beliebtheit. Damit wächst auf Seiten der Entwickler der Bedarf an Biliotheken, welche die Entwicklung solcher Systeme unterstützen und erleichtern. Bei der Spezifikation eines Kommunikationsprotokolles müssen die Eigenschaften der zugrundeliegenden Netzwerkprotokolle berücksichtigt werden, sowohl in der Design- als auch in der Entwicklungsphase. Bibliotheken helfen den Entwicklern, ihr Kommunikationsprotokoll mehr oder minder unabhängig vom Medium zu implementieren. Außerdem sollten dem Entwickler high-level Konzepte zur Fehlerbehandlung zur Verfügung stehen.

Port-based Distributed Haskell [1] [2] bietet solch eine Bibliothek für die funktionale Programmiersprache `Haskell` [4]. Das Laufzeitsystem stellt der Anwendung sogenannte Ports zur Verfügung, über die ähnlich wie über Kanäle in Concurrent Haskell [5] getypte Nachrichten in Form von algebraischen Datentypen versendet werden können. Diese Nachrichten werden mittels eines Netzwerkprotokolles (hier über TCP) zwischen den einzelnen Rechnern ausgetauscht. Dabei ist es notwendig, die Daten von der internen Darstellung in Haskell in einen Strom von Bytes zu konvertieren. Auf der Gegenseite muß diese Umwandlung wieder rückgängig gemacht werden. Für grundlegende Datentypen eignet sich Haskells `Show` bzw. `Read`-Klasse. Da Haskell stark getypt ist, müssen einige Besonderheiten in der Implementierung beachtet werden. Außerdem sind zur effizienten Verwendung der Bibliothek Optimierungen vor allem auf der Netzwerkebene wichtig.

2 Grundlagen und Concurrent Haskell

Bei der Entwicklung eines verteilten Systems in Haskell stößt man schnell an die Grenzen des mit den Standardbibliotheken Möglichen. Haskell verfügt über eine Bibliothek zur nebenläufigen Programmierung in Concurrent Haskell und einer weiteren zur Netzwerkprogrammierung notwendigen Bibliothek. Letztere bietet nur die

Möglichkeit, einen Zeichenstrom als `String` zu übertragen. Wir werden zeigen, wie aus beiden Bestandteilen eine wertvolle Haskell-Bibliothek entsteht.

Wenn wir den Aspekt der Verteiltheit außer Acht lassen, eignen sich die Kanäle aus Concurrent Haskell als Implementierung eines Ports über einem algebraischen Datentyp `a`: Sie bieten eine FIFO-Queue, in die mittels `writeChan :: Chan a -> IO ()` Daten geschrieben und mit `readChan : Chan a -> IO a` wieder ausgelesen werden können. Dabei können durchaus mehrere schreibende Prozesse vorhanden sein. Sollte ein Prozeß aus einem leeren Kanal lesen, wird er solange suspendiert, bis ein anderer Prozeß in diesen Kanal hineinschreibt. Neue Prozesse können mit der Funktion `forkIO` erzeugt werden. Es ist zu beachten, daß alle erzeugten Prozesse abgesehen von der Synchronisation über die Kanäle nebenläufig arbeiten. Wir übernehmen diese Aktionen als `newPort`, `readPort` und `writePort` in unsere Bibliothek.

Eine einfache Client/Server-Anwendung könnte beispielsweise so aussehen:

```
main :: IO ()
main = do
  p <- newPort
  forkIO (writer p 1)
  reader p
```

```
reader p = do
  i <- readPort p
  putStrLn "read:" ++ (show i)
  reader p
```

```
writer p i = do
  writePort p i
  writer p (i+1)
```

Zwei Ports können mit `mergePort :: Port a -> Port b -> IO (Port (Either a b))` verschmolzen werden. Durch eine einfache Fallunterscheidung läßt sich dann feststellen, von welchem Port die gelesene Nachricht kam.

Bei einem Zugriff über das Netzwerk muß der Port, an den eine Nachricht gesandt werden soll, anders spezifiziert werden: Wir möchten dazu den Rechnernamen und einen symbolischen Dienstnamen verwenden. Deshalb stellt Port-basiertes Distributed Haskell folgende Funktionen zur Verfügung:

```
registerPort :: Port a   -> Portname -> IO ()
lookupPort   :: Hostname -> Portname -> IO (Port a)
```

Zusätzlich zu den Kommunikationsprimitiven bietet die Bibliothek zwei Funktionen zur robusten Programmierung: Mittels `link :: Port a -> IO () -> IO Link` können beliebige IO-Aktionen ausgeführt werden, falls der im ersten Parameter angegebene Port nicht mehr erreichbar ist oder eine Nachricht nicht zustellbar war (Fehler im Netzwerk, aufgelegtes Modem). Das Laufzeitsystem überwacht diese gelinkten Ports durch periodisches Polling, bis der Link mit der Funktion `unlink :: Link -> IO ()` wieder abgebaut wird.

3 Das Laufzeitsystem

Wie schon erwähnt liegen der internen Kommunikation vor allem die Kanäle Concurrent Haskells zugrunde. Eine weitere wichtige Komponente ist die Netzwerkschicht. Für jede zu verschickende Nachricht muß der Benutzer Instanzen der Klasse `Serialize` implementieren, welche mittels der Funktionen `serialize/deserialize` die Konvertierung der Haskell-Objekte in einen Bytestrom gewährleisten. Für Basistypen und algebraische Datentypen kann ohne zusätzlichen Aufwand die Instantiierung über die Klassen `Read` und `Show` gewählt werden, welche direkt seitens Haskell zur Verfügung stehen. Lediglich für komplexere Datenstrukturen wie beispielsweise Graphen ist eine eiffizientere Implementierung des Marshallings von Haskell in einen Bytestrom und umgekehrt empfehlenswert. Beim Senden an einen entfernten Port werden diese Daten dann zusammen mit der Information, für welchen Port sie bestimmt sind, über das Netz übertragen.

Leider führt der intuitive Ansatz, für jeden Port einen getypten Kanal zu nehmen, zu einem Konflikt mit dem Haskell-Typsystem: Auf der annehmenden Seite, im sogenannten `PortListener` würde dies dazu führen, daß diese Funktion unter anderem eine Datenstruktur, verwalten müßte, welche getypt ist über alle auf diesem Knoten vorhandenen Ports resp. Kanäle. Da diese aber alle über unterschiedliche Typen (z.B. algebraischen Typen, `Strings`, `Int`) gebildet sind und sich somit nicht in einem einzigen Typen zusammenfassen lassen, müssen wir zu einen Weg finden: Jedem internen Port ordnen wir nicht nur seinen getypten Kanal, sondern auch einen Kanal vom Typ `String` zu, also genau dem Datenformat, indem wir die noch nicht mit `deserialize` bearbeiteten Nachrichten erhalten. Zusätzlich starten wir pro Port einen neuen Prozeß, der in einer Schleife Daten aus dem Kanal vom Typ `String` liest, mit `deserialize` konvertiert und dann in den endgültigen Kanal schreibt. Der `PortListener` braucht somit nur alle Kanäle des Typs `String` zu verwalten, was sich ohne Probleme mit dem Typsystem Haskells bewältigen läßt. Es ergibt sich somit der in Abbildung 1 gezeigte Verlauf einer eingehenden Nachricht bis hin zur `readPort`-Anweisung.

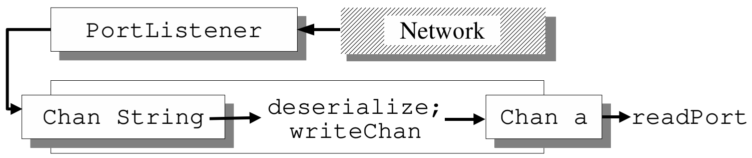


Abbildung 1. Über das Netzwerk eingehende Nachricht

Bei der Kommunikation über TCP wird zwischen zwei kommunizierenden Anwendungen eine einzige Verbindung aufgebaut, über die der gesamte Nachrichtenverkehr abgewickelt wird. Diese Optimierung ist wichtig, da der Verbindungsaufbau von TCP sehr zeitaufwendig ist und es somit zu Zeitverzögerungen bei vielen Nachrichten hintereinander kommt. Desweiteren läßt sich bei vielen parallelen Nachrichten auch nur eine bestimmte Anzahl von Verbindungen benutzen, da das Betriebssystem nur begrenzte Ressourcen für deren Verwaltung zur Verfügung hat.

Außer den Nachrichten an einen Port werden über diese auch beispielsweise die Kontrollinformationen für das Linking und den Namensdienst übertragen. Um die verschiedenen Nachrichten auf der Seite des Empfängers wieder an das entsprechende Subsystem weiterleiten zu können, werden die Nachrichten vor dem Versenden mit einem Präfix versehen. Im `PortListener` wird auf der anderen Seite wie in Abbildung 2 dargestellt das Präfix abgestreift und die Nachrichten über eine generische Routine an das Laufzeitsystem übergeben.

4 Streams

Der bisherige Ansatz zur Kommunikation in der IO-Monade [3] hat jedoch einige Nachteile. Zum einen lassen sich gewisse Beschränkungen nicht durch Typsignaturen

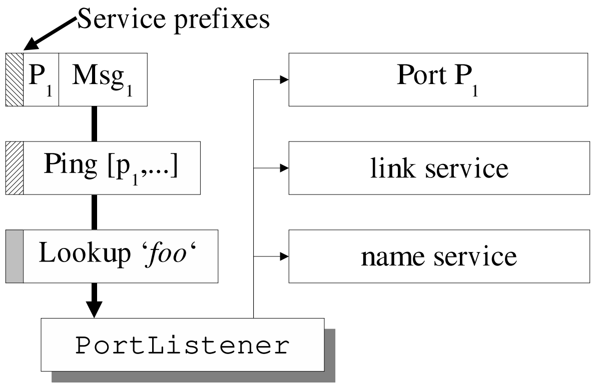


Abbildung 2. Verteilung der Nachrichten an die Subsysteme

formulieren. Zum anderen gehen Vorteile der funktionalen Programmierung verloren. Beispielsweise bietet es sich für das Bearbeiten eingehender Nachrichten an, diese in einer (unendlichen) Liste zu speichern und mit funktionalen Mitteln wie `fold` und `map` abzuarbeiten.

Genau dies ist mit *Streams* möglich. Statt einem Port bietet die Bibliothek mit `newStream :: IO (Port a, [a])` einen unendlichen Strom der eingehenden Nachrichten an diesem Port. Die explizite Referenz auf den Port ist weiterhin für Sendeoperation nötig, außerdem wird sie von den Befehlen `register` und `link` benutzt. Als Ersatz für `mergePort` wird nun `mergeStreams :: [a] -> [b] -> IO [Either a b]` verwendet. Eine herkömmliche Anwendung mit expliziten `readPort`-Anweisungen kann nun in einer wesentlich präziseren Form angegeben werden. Als Beispiel dient ein Fragment aus einem `Chat`-Server, bei dem ein Client zwei Prozesse/Ports benutzt, um Nachrichten vom Server und die Eingaben des Benutzers an der Tastatur zu behandeln.

```
server = do
 (p,stream) <- newStream
 register p "Server"
 foldM_ work
    initialState stream
 where
  work st (Connect him) = ..
  work st (Close   him) = ..
```

```
client = do
  server <- lookup host "Server"
  (me,s) <- newStream
  writePort server (Connect me)
  (kport,kstream) <- newStream
  forkIO (readKeyboard kport)
  stream <- mergeStreams kstream s
  mapM (loop server) $
    takeWhile (/= (Left "")) stream
  writePort server (Close me)
```

5 Zusammenfassung

In dieser Arbeit haben wir einen kleinen Überblick über die Verwendung von Port-basiertem Distributed Haskell zur Programmierung eines robusten verteilten Systems in Haskell gegeben. Aufbauend auf Concurrent Haskell bietet die Bibliothek Ports, über die Nachrichten verschickt werden können. Konstrukte zur robusten Programmierung stehen zur Verfügung. Einige Besonderheiten der Implementierung wurden betrachtet. Abschließend wurde die Erweiterung der Bibliothek um Streams vorgestellt, so daß sich die Kommunikation auch wieder in einem funktionalen Kontext mittels Strömen von Nachrichten betrachten läßt.

Im Gegensatz zu vielen anderen Implementierung von Bibliotheken zur verteilten Programmierung in Haskell ist unsere Bibliothek keine Erweiterung eines speziellen Compilers wie zum Beispiel Glasgow parallel oder distributed Haskell. Die Bibliothek läßt sich unabhängig von der Entwicklung des Compilers warten und weiterentwickeln. So können von den Anwendern jeweils die Neuerungen in der Entwicklung von Haskell direkt übernommen werden, ohne lange auf eine Portierung des erweiterten Systems warten zu müssen.

Zur Zeit benötigt Port-basiertes Distributed Haskell den Glasgow Haskell Compiler `ghc` ab Version 4.08.1 oder höher. Die Bibliothek ist verfügbar unter `http://www-i2.informatik.rwth-aachen.de/Research/distributedHaskell/`.

Literatur

1. F. Huch and U. Norbisrath. Distributed Programming in Haskell with Ports. *LNCS*, 2011, 2000.
2. Frank Huch and Volker Stolz. Implementation of Portbased Distributed Haskell. In Thomas Arts and Markus Mohnen, editors, *IFL 2001*, September 2001.
3. S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. `http://research.microsoft.com/\char126simonpj/\#marktoberdorf`, January 2001.
4. S. Peyton Jones et al. Haskell 98 report. Technical report, `http://www.haskell.org/`, 1998.
5. J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. D. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Peyton Jones, A. Reid, and P. Wadler. Report on the Programming Language Haskell (Version 1.4), April 1997.

WASH/CGI: Server-side Web Scripting with Sessions, Compositional Forms, and Graphics

Peter Thiemann

Universität Freiburg
thiemann@uni-freiburg.de

Abstract The common gateway interface (CGI) is one of the prevalent methods to provide dynamic contents on the Web. Since it is cumbersome to use in its raw form, there are many libraries that make CGI programming easier.

WASH/CGI is a domain specific embedded language for server-side Web scripting. It is implemented and hosted in Haskell. Its implementation relies on CGI, but it avoids most of CGI's drawbacks by incorporating the concept of a session and by providing a compositional approach to constructing interaction elements (forms). From a programmer's perspective, programming WASH/CGI is like programming a graphical user interface (GUI). In contrast to a GUI, the layout is specified using HTML. WASH/CGI generates HTML via a new monadic interface. Special combinators are available that provide typed input fields and graphics, which is generated on the fly.

1 Introduction

The common gateway interface (CGI) is one of the oldest methods for deploying dynamic Web pages based on server-side computations. As such, CGI has a number of advantages. Virtually every Web server supports CGI. CGI requires no special functionality from the browser, apart from the standard support for form elements in HTML. On the programming side, CGI communicates via standard input/output streams and environment variables. It is not tied to a particular architecture or implementation language. Hence, CGI is the most portable approach to providing dynamic contents on the Web.

The basic idea of CGI is straightforward. Whenever the Web server receives a request for a CGI-enabled URL, it treats the local file determined by the URL as an executable program and starts it in a new process. This kind of program is called a *CGI script*. It receives its input through the standard input stream and through environment variables and delivers the response to its standard output stream. The CGI standard [4] fixes the format of this communication.

Unfortunately, there are a number of limitations. The most painful one stems from the fact that the underlying HTTP protocol, which is used for communication between browser and server, is *stateless*. Every single request starts a CGI script. Then the script produces a response page and terminates. Hence, there is no concept of a *session*, *i.e.*, a sequence of alternating requests and responses. Usually, CGI programmers must build such sessions from scratch. They distribute the stages of a session over a number of CGI scripts and connect them manually through links in the response pages. To provide a notion of session-wise state they must resort to putting hidden information in their responses (hidden input fields) or to using cookies, which is not reliable because browsers can refuse them. Clearly, it is error-prone to manually maintain links in this way and also to have the code for a single interaction forcibly spread over many programs.

Another source of errors lies in the parameter passing scheme between forms and CGI scripts. A form is an HTML element that contains named input elements. Each input element implements one particular kind of input behavior (a widget in GUI terminology). When a special submit button is pressed, the browser sends a list of pairs of input element names and their string values to the server. Inside of a CGI script, these argument values can be accessed by their name. Unfortunately, there is no guarantee that the form uses the names expected by the script and vice versa.

Last but not least, all parameter passing between forms and CGI scripts is completely untyped. Each script must provide its own decoding functions to convert strings into whatever type is really required. It is not even possible to specify the expected type of an input field.

The present work provides a cure for all the issues mentioned above: the DSL WASH/CGI. WASH/CGI makes CGI programming easy and intuitive. It is implemented as a library for Haskell [7] and provides the following features:

- one program can implement entire sessions;
- the specification of an input field and the collection of the input from this widget are tied together so that mismatches are not possible; the external name of an input field does not matter;
- input fields are first-class entities; they may be typed and grouped to compound input fields (compositionality); each group may be bound to a callback action;
- first-class images as active input fields where each pixel of the image can result in a different action;
- no explicit URLs need to be constructed in the script, except references to external pages;
- the script is "relocatable"; it can be moved in the directory hierarchy or to another server without change[1].

The library is available through the WASH web page [17]. The web page also provides some live examples, complete with sources. The implementation of WASH is documented elsewhere [14].

Familiarity with the Haskell language [7] as well as with the essential HTML elements is assumed throughout.

2 Example programs

This section demonstrates the use of WASH/CGI with some examples. At first, the reader may be surprised that the examples have a distinct GUI flavor. But this is exactly the right impression: CGI programming should feel just like GUI programming, where the layout is determined by HTML.

The library is based on the monad `CGI`, which handles all interaction with the browser.

2.1 Hello world

```
mainCGI :: CGI ()
mainCGI =
  htell (standardPage "Hello World" empty)
```

[1] Of course, provided that it can execute at all on the other machine.

```
mainCGI :: CGI ()
mainCGI =
  counter 0

counter :: Int -> CGI ()
counter n =
  ask (standardPage "Counter" $
       makeForm $
       do text "Current counter value "
          text (show n)
          br empty
          submitField (counter (n + 1))
                      (fieldVALUE "Increment"))
```

Figure 1. The counter example

As customary, the first program just displays `Hello World` on the screen. The combinator `htell` takes an HTML page produced by `standardPage` and sends it to the browser. The combinator `standardPage` takes a title and the contents of the HTML page (here: `empty`) and produces the usual combination of `html`, `head`, `body`, and `title` tags.

```
standardPage ttl elems =
  html (head (title (text ttl)) ##
        body (h1 (text ttl) ## elems))
```

Here, `text` transforms a string into an HTML element and the operator `##` concatenates groups of HTML elements. Hence, the browser receives the following response:

```
<html><head><title>Hello World</title>
</head>
<body><h1>Hello World</h1>
</body>
</html>
```

2.2 The counter example

The counter example (Fig. 1) uses the `makeForm` combinator to start a form. The contents of the form are specified using a monad. Every content element (in fact, every HTML element) is a value in the monad `WithHTML CGI`. The `text` combinator produces plain text output, the `br empty` inserts a `<br>` element, and the `submitField` creates a submit-button. The first parameter of `submitField` is the action to be taken, when the form is submitted. The second parameter specifies the attributes of the submit-button. The `empty` parameter of `br` can also be replaced by attributes for `<br>`.

2.3 Extended counter

```
counter :: Int -> CGI ()
counter n =
```

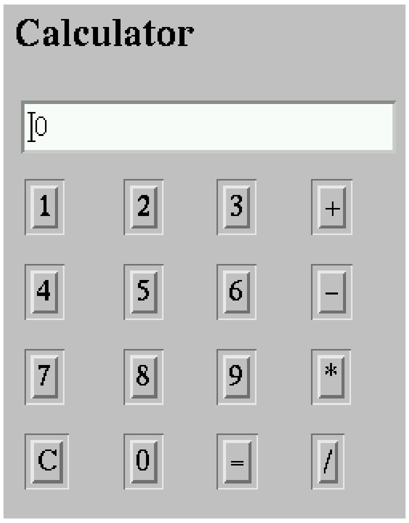


Figure 2. Screenshot of Calculator

```
ask $ standardPage "UpDownCounter" $ makeForm $
do text "Current counter value "
   activeInputField counter (fieldVALUE (show n))
   submitField (counter (n + 1)) (fieldVALUE "++")
   submitField (counter (n - 1)) (fieldVALUE "--")
```

In this example, we consider a replacement for the `counter` function from the previous example. It displays the current value in an input field and it has two submit-buttons. The generated Web page has the following functionality: Clicking on the `++` and `--` button increments or decrements the counter's value. Alternatively, a new value may be entered by typing it into the `activeInputField` and hitting return to submit the form. The input field is actually typed. It accepts only inputs that parse as elements of type `Int`:

```
activeInputField :: Read a =>
        (a -> CGI ()) -> HTMLField (InputField a)
```

2.4 Calculator

A pocket calculator consists of a display and an array of buttons (Fig. 2). The corresponding code uses a HTML table to specify the layout (Fig. 3). Each button has an action, specified by `calcAction`, attached to it. The `textInputField` is a specialized input field for Strings. Including it into a form yields a value `dsp` of type `InputField String`. This value is a handle to extract the input from the field using the function `value :: InputField a -> Maybe a`.

The `table`, `tr`, and `td` functions construct HTML elements with the same tag. The argument of each function specifies the list of sub-elements and attributes of the element. The operator `##` serves to concatenate (lists of) HTML elements and attributes. In the example code,

```
mainCGI :: CGI ()
mainCGI =
  calc "0" id

calc :: String -> (Integer -> Integer) -> CGI ()
calc dstr f =
  ask $ standardPage "Calculator" $ makeForm $ table $
  do dsp <- tr (td (textInputField (fieldVALUE dstr)
                    ## attr "colspan" "4"))
     let btn c = td (submitField (calcAction dsp c f)
                                (fieldVALUE [c]))
     tr (btn '1' ## btn '2' ## btn '3' ## btn '+')
     tr (btn '4' ## btn '5' ## btn '6' ## btn '-')
     tr (btn '7' ## btn '8' ## btn '9' ## btn '*')
     tr (btn 'C' ## btn '0' ## btn '=' ## btn '/')

calcAction :: InputField String ->
              Char -> (Integer -> Integer) -> CGI ()
calcAction dsp c f
  | isDigit c = calc (dstr ++ [c]) f
  | c == 'C'  = mainCGI
  | c == '='  =
       calc (show (f (read dstr :: Integer))) id
  | otherwise =
       calc "0" (optable c (read dstr :: Integer))
  where Just dstr = value dsp
        optable '+' = (+)
        optable '-' = (-)
        optable '*' = (*)
        optable '/' = div
```

Figure 3. Calculator

```
td (textInputField (fieldVALUE dstr) ##
    attr "colspan" "4")
```

creates the element

```
<td colspan="4"> <input type="text" value="..."> </td>
```

2.5 Graphics

```
mainCGI =
  ask $ standardPage "UseGraphics" $ makeForm $
  activeImage testImage

canvasRed = newImage (100,100) red
ovalBlue = fillOval canvasRed (20,20) (70,50) blue
background = activate ovalBlue hitNothing
testImage = activateColor background blue hitOval

hitOval = htell (standardPage "Hit the Oval!" empty)
```

```
hitNothing = htell (standardPage "Missed." empty)

red  = (255,0,0)
blue = (0,0,255)
```

The connection to GUIs also extends to images where parts of the image may be bound to certain actions. The example program constructs a red square (`canvasRed`) and paints a blue oval in it (`ovalBlue`). Next it activates the image, so that an action is triggered when the image is clicked. The image `background` executes the action `hitNothing` everywhere. The image `testImage` has the action `hitOval` attached to each blue pixel in `background`, and the action `hitNothing` to any other pixel.

3 User-level concepts

This section presents an application programmer's view of the concepts and function of WASH/CGI.

3.1 HTML

Each HTML element is constructed by a function of the same name as shown with the function `table` below. Each of these "constructor functions" has a type like

```
type HTMLCons m a = WithHTML m a -> WithHTML m a
table :: Monad m => HTMLCons m a
```

There are also constructor functions for attributes that will be attached *to the enclosing element*. The generic attribute constructor is `attr`.

```
attr :: Monad m => String -> String -> WithHTML m ()
```

It constructs an attribute instance from an attribute name and an attribute value.

Although a value of type `WithHTML m a` stands for an ordered collection of HTML elements and attributes, it is impossible to examine elements and attributes once they are constructed. Passing such a value to a constructor function incorporates the elements as sub-elements of the new element and also attaches the attributes to it.

The values

```
empty :: Monad m => WithHTML m ()
(##)  :: Monad m => WithHTML m a ->
                    WithHTML m b -> WithHTML m a
```

serve as the empty collection and as the concatenation operation. Since `WithHTML m` is a monad (provided that `m` is), HTML elements may also be combined using the standard monad operations as well as the `do` notation.

In most cases, the parameter `m` will be the monad `CGI`. But there are exceptions, as we will see below in 3.4.

```
passwordInputField :: HTMLField (InputField String)
checkboxInputField :: HTMLField (InputField Bool)
fileInputField     :: HTMLField (InputField String)
resetField         :: HTMLField (InputField ())
submitField        :: CGI () -> HTMLField ()
```

Figure 4. Input fields (excerpt)

3.2 Input fields

There are special combinators to construct input fields. They add an input field to the current collection of HTML elements and return a handle for accessing the input value. The type of such an input field is

```
type HTMLField a = WithHTML CGI () -> WithHTML CGI a
```

That is, it takes a collection of attributes (of type `WithHTML CGI ()`), attaches them to a new `<input>` field, and embeds the new field into another collection. The generic, typed textual input field is constructed by

```
inputField :: Read a => HTMLField (InputField a)
```

The `Read a` predicate comes from the fact that all communication between browser and server is through strings. Hence, each value of type `a` must be converted from a string.

Once again, the value of type `InputField a` is merely a handle to access its input values through the two functions

```
value  :: InputField a -> Maybe a
string :: InputField a -> Maybe String
```

The `value` function provides access to the parsed value (if there was a parsable input), whereas the `string` function is meant for error analysis and provides access to the raw input (if the input element was filled in at all).

The remaining input elements are provided in the same manner (see Fig. 4). A `fileInputField` returns the contents of the chosen file as a string. A `resetField` just clears all input fields, it has no I/O functionality. Radio buttons and selection boxes have a slightly more complicated interface. They are omitted for brevity.

It remains to discuss the `submitField`. It takes a `CGI` action and generates a button in the HTML page. Clicking such a button executes its action. The action is similar to a continuation. Since a form may contain more than one submit button, multiple continuations are possible. In particular, a large form may be composed from small interaction groups that consist of input fields and one or more submit buttons.

3.3 Forms

```
makeForm :: HTMLField ()
```

The constructor for forms takes a collection of attributes and returns a `<form>` element. At least one form is necessary in each page that contains input fields since input fields do not make sense outside of a form. It is not necessary to set the standard attributes of the form element. The `action` attribute, which contains the URL for processing the form's contents, the `enctype` attribute, which determines the encoding of the form's contents, and the `method` attribute are all determined automatically by WASH/CGI.

3.4 Sessions

Programming of interactions is based on just four combinators in the monad `CGI`.

```
run  :: Translation -> CGI () -> IO ()
ask  :: WithHTML CGI a -> CGI ()
tell :: CGIOutput a => a -> CGI ()
io   :: (Read a, Show a) => IO a -> CGI a
```

The combinator `run` introduces the `CGI` monad. The standard main program (for the examples above) is as follows:

```
main = run NoTranslation mainCGI
```

The `NoTranslation` argument is appropriate for all scripts that do not make use of graphics generation (see Sec. 3.5).

The combinator `ask` displays a page on the browser. Its argument of type `WithHTML CGI a` constructs a web page, which should contain a form. It returns a `CGI` action. This action never produces its value. To extract values from the form, a callback action must be tied to one of its input elements.

The combinator `tell` displays a page and terminates the interaction. The argument of `tell` can have any type of class `CGIOutput`. These types are `Status` (error message), `Location` (redirect response to a URL), and `WithHTML IO a` (a HTML page). Actually, `tell` is a member function of `CGIOutput`.

```
class CGIOutput a where
  tell :: a -> CGI ()
  cgiPut :: a -> IO ()
```

The combinator `io` injects an `IO` action into the `CGI` monad.

3.5 Graphics

Many web pages contain graphics that are prefabricated. The WASH/CGI library contains facilities to create simple, click-sensitive graphics on the fly. As expected from a functional language, an image is a first-class value. Images can be created from scratch (`newImage`), from text strings (`makeText`), and from existing GIF images (`gifImage`). They can be manipulated in the usual ways by drawing ovals, rectangles, and lines and by overlaying one image on top of another (see Figure 5 for a summary of the interface). In addition, `CGI` actions may be attached to parts of an image using `activate`, `activateXY`, and `activateColor`. Finally, an image can be inserted into a web page using `activeImage`. In principle, it is possible to write interactive graphics programs, although the interaction is a bit slow.

```
-- inserting into HTML
activeImage   :: CGIImage -> WithHTML CGI ()
-- creating new images
newImage      :: (Int, Int) -> Pixel -> CGIImage
makeText      :: String -> Pixel -> CGIImage
gifImage      :: String -> CGIImage
-- drawing
drawOval, fillOval, drawRectangle, fillRectangle,
  drawLine    :: CGIImage -> (Int, Int) -> (Int, Int) ->
                 Pixel -> CGIImage
-- composing images
overlay       :: CGIImage -> CGIImage -> (Int, Int) ->
                 Pixel -> CGIImage
-- attaching actions
type ActionFun = Int -> Int -> Maybe (CGI ())
activateXY    :: CGIImage -> ActionFun -> CGIImage
activate      :: CGIImage -> CGI () -> CGIImage
activateColor :: CGIImage -> Pixel -> CGI () -> CGIImage
```

Figure 5. Interface for graphics

4 Related work

Meijer's CGI library [10] implements a low-level facility for accessing the input to a CGI script and for creating its output. It is nicely engineered and its functionality is at about the level of our own `RawCGI` library. However, Meijer's library offers additional features like cookies and its own HTML representation, which we felt should be separated from the functionality of `RawCGI`.

Hughes [9] has devised the powerful concept of arrows, a generalization of monads. His motivating application is the design of a CGI library that implements sessions. Indeed, the functionality of his library was the major source of inspiration for our work. Our work indicates that monads are sufficient to implement sessions (Hughes also realized that [8]). Furthermore, it extends the functionality offered by the arrow CGI-library with a novel representation of HTML, compositional forms, and graphics. Also, the callback-style of programming advocated here is not encouraged by the arrow library.

Hanus's library [6] for server-side scripting in the functional-logic language Curry comes close to the functionality that we offer. In particular, its design inspired our switching to a callback-style programming model. While his library uses logical variables to identify input fields in HTML forms, we are able to make do with a purely functional approach. Our approach only relies on the concept of a monad, which is fundamental for a real-world functional programmer.

Bigwig [13] is a system for writing Web applications. It provides a number of domain specific customizable languages for composing dynamic documents, specifying interactions, accessing databases, etc. It compiles these languages into a combination of standard Web technologies, like HTML, CGI, applets, JavaScript. Like our library, it implements a session facility, which is more restrictive in that sessions may neither be backtracked nor forked. Each Bigwig session has a notion of a current state, which cannot be subverted. However, the implementation of sessions is different and relies on a special runtime system that improves the efficiency of CGI scripts [2].

In addition, Bigwig provides a sophisticated facility for generating documents and typed document templates. Moreover, there is a type system for forms. WASH/CGI provides typed document template in the weak sense of Bigwig by keeping strings and values of type `Element` apart. A special type system for forms is not required since (typed) field values are directly passed to (typed) callback-actions. Hence, all necessary type checking is done by the Haskell compiler.

MAWL [1] is a domain specific language for specifying form-based interactions. It provides a subset of Bigwig's functionality, but it was the first language to offer a typing of forms against the code that received its input from the form. In particular, the facilities for document templates are much more limited.

In comparison to a GUI library [3, 5, 12, 15] a CGI library does not have to deal with concurrency. All interaction is limited to exchanging messages between Web browser and Web server, so that nested interactions are not possible. This greatly simplifies the implementation. However, HTML is an expressive language to specify layout, even for a GUI, and Web-based user interfaces are ubiquitous, so there is a market for the kind of library that we are proposing.

5 Conclusions

The WASH/CGI library brings new power to CGI programmers. It offers an easy and declarative way to implement complicated interactive Web-based user interfaces. In particular, it treats the display of the Web browser like a graphical user interface with restricted facilities. This approach results in a natural use of HTML for the layout and in the use of callback-actions to specify the flow of control.

The WASH/CGI approach is not only suitable for CGI programming, but also for other kinds of server-side Web scripting. For example, it would be interesting to investigate a combination with Haskell server pages [11], with Bigwig's runtime system [2], or with proprietary APIs.

References

1. David Atkinson, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, and Kenneth Rehor. Experience with a domain specific language for form-based services. In *Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997. USENIX.
2. Claus Brabrand, Anders Møller, Anders Sandholm, and Michael I. Schwartzbach. A runtime system for interactive web services. *Journal of Computer Networks*, 1999.
3. Magnus Carlsson and Thomas Hallgren. FUDGETS: A graphical interface in a lazy functional language. In Arvind, editor, *Proc. Functional Programming Languages and Computer Architecture 1993*, pages 321–330, Copenhagen, Denmark, June 1993. ACM Press, New York.
4. Cgi: Common gateway interface. `http://www.w3.org/CGI/`.
5. Sigbjørn Finne and Simon Peyton Jones. Composing Haggis. In *Proc. 5th Eurographics Workshop on Programming Paradigms in Graphics*, Maastricht, NL, September 1995.
6. Michael Hanus. High-level server side Web scripting in Curry. In *Practical Aspects of Declarative Languages, Proceedings of the Third International Workshop, PADL'01*, Lecture Notes in Computer Science, Las Vegas, NV, USA, 2001. Springer-Verlag.
7. Haskell 98, a non-strict, purely functional language. `http://www.haskell.org/definition`, December 1998.

8. John Hughes. Private communication, September 2000.
9. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
10. Erik Meijer. Server-side web scripting with Haskell. *Journal of Functional Programming*, 10(1):1–18, January 2000.
11. Erik Meijer and Danny van Velzen. Haskell Server Pages, functional programming and the battle for the middle tier. In *Draft proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, pages 23–33, Montreal, Canada, September 2000.
12. Meurig Sage. FranTk — a declarative GUI language for Haskell. In Wadler [16], pages 106–117.
13. Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic web documents. In Tom Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 290–301, Boston, MA, USA, January 2000. ACM Press.
14. Peter Thiemann. Wash/CGI: Server-side Web scripting with sessions and typed, compositional forms. Lecture Notes in Computer Science, Portland, OR, USA, January 2002. Springer-Verlag.
15. Ton Vullinghs, Daniel Tuijnman, and Wolfram Schulte. Lightweight GUIs for functional programming. In Doaitse Swierstra and Manuel Hermenegildo, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95)*, number 982 in Lecture Notes in Computer Science, pages 341–356, Utrecht, The Netherlands, September 1995. Springer-Verlag.
16. Philip Wadler, editor. *International Conference on Functional Programming*, Montreal, Canada, September 2000. ACM Press, New York.
17. Web authoring system in Haskell (WASH). `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH`, March 2001.

How to Integrate Declarative Languages and Constraint Systems

Petra Hofstedt

Technische Universität Berlin
ph@cs.tu-berlin.de

Abstract This paper shortly describes a general approach for the integration of arbitrary declarative languages and constraint systems. Our approach allows to build constraint languages according to current requirements and, thus, comfortable modelling and solving of a wide range of problems.

1 Introduction

While constraint languages usually are extensions of an initial programming language with constraints, where the evaluation mechanism of the initial language is extended by some special rules for handling constraints, our point of view is different from that. In our approach we consider a system of cooperating constraint solvers [2, 3] which initially is not equipped with a certain programming language. However, the system allows the integration of different host languages by treating them as constraint solvers. This enables to build constraint languages according to current requirements.

2 Constraint Programming and Constraint Solvers

A *signature* $\Sigma = (S, F, R; ar)$ consists of a set S of sorts, a set F of function symbols, a set R of predicate symbols, and an arity function $ar : F \cup R \to S^{\star}$. A set of variables appropriate to Σ is a many sorted set $X = \bigcup_{s \in S} X^s$, where $\forall s \in S$ the set X^s is countably infinite. A *Σ-structure* $\mathcal{D} = (\{\mathcal{D}^s \mid s \in S\}, \{f^{\mathcal{D}} \mid f \in F\}, \{r^{\mathcal{D}} \mid r \in R\})$ consists of an S-sorted family of nonempty carrier sets $\mathcal{D}^s$, a family of functions $f^{\mathcal{D}}$, and a family of predicates $r^{\mathcal{D}}$ appropriate to F and R. Let the set of terms $\mathcal{T}(F, X)$ be defined as usually.

Let $\Sigma = (S, F, R; ar)$ be a signature, where R contains at least one predicate symbol $=^s_\Sigma$ for every $s \in S$. Let X be a set of Σ-variables. Let $\mathcal{D}$ be a Σ-structure with equality, i.e. for every predicate symbol $=^s_\Sigma$ there is a predicate $=^s_{\mathcal{D}} \subseteq \mathcal{D}^s \times \mathcal{D}^s$, for which the usual axioms for equality hold. A *constraint* is a string $r(t_1, \ldots, t_m)$, where $r \in R$ with $ar(r) = s_1 \ldots s_m$ and $t_i \in \mathcal{T}(F, X)^{s_i}$. The set of constraints over Σ is denoted by $\mathcal{C}onstraint$. It contains, furthermore, the two distinct constraints $true$ and $false$ with $\mathcal{D} \vDash true$ and $\mathcal{D} \nvDash false$. The 4-tupel $\zeta = (\Sigma, \mathcal{D}, X, \mathcal{C}ons)$, where $\{true, false\} \subseteq \mathcal{C}ons \subseteq \mathcal{C}onstraint$, is a *constraint system*.

A *solution* of a disjunction C of constraint conjunctions in $\mathcal{D}$ is a valuation $\sigma : Y \to \mathcal{D}$, where $var(C) \subseteq Y \subseteq X$, such that $(\mathcal{D}, \sigma) \vDash C$ holds. *Solving* the disjunction C means finding out whether there is a solution for C or not, i.e. finding out whether C is satisfiable in $\mathcal{D}$ or not.

Given a constraint system, we need appropriate algorithms for constraint manipulation. A *constraint solver* CS is associated with a constraint system ζ. It is a collection of operations on *disjunctive constraints,* i.e disjunctions of constraint conjunctions, of the associated constraint system. Typically a constraint solver consists of a combination of instantiations of the operations constraint satisfaction, constraint entailment, projection and simplification.

3 Cooperating Constraint Solvers

While the paradigm of constraint programming offers efficient mechanisms to handle constraints of various constraint domains, it has been shown to be desirable to combine several constraint solving techniques because this combination makes it possible to solve problems that none of the single solvers can handle alone. Thus, in [2,3] we introduced a flexible combination mechanism for constraint solvers. Moreover, using this mechanism it is possible to integrate different host languages into the system by considering their evaluation mechanisms as constraint solvers. In this section, we shortly recall our combination mechanism.

Since we want to solve mixed disjunctive constraints such that every constraint may contain function symbols and predicate symbols of different constraint systems it is necessary to convert every such disjunction into a disjunction such that every constraint is defined by function symbols and predicate symbols of exactly one constraint system. This is done by flattening. In [3] we give a definition of the function $Flatten$ and we show that the set of solutions w.r.t. the common variables is preserved. Thus, after flattening we can solve the newly built disjunction instead of the original mixed one.

Figure 1 shows the architecture of our overall system for cooperating solvers. Let L be the set of indices of constraint systems, $\mu, \nu \in L$. To every individual solver CS_ν a *constraint store* C^ν is assigned. Let $\mathcal{DC}Cons_\nu$ denote the set of disjunctive constraints of ζ_ν. A constraint store $C^\nu \in \mathcal{DC}Store_\nu \subseteq \mathcal{DC}Cons_\nu$ is a disjunctive constraint which is satisfiable in the corresponding structure. The *meta constraint solver* coordinates the work of the different individual solvers and it manages the *constraint pool.* Initially, the constraint pool contains the constraints of the constraint conjunction Φ which we want to solve.

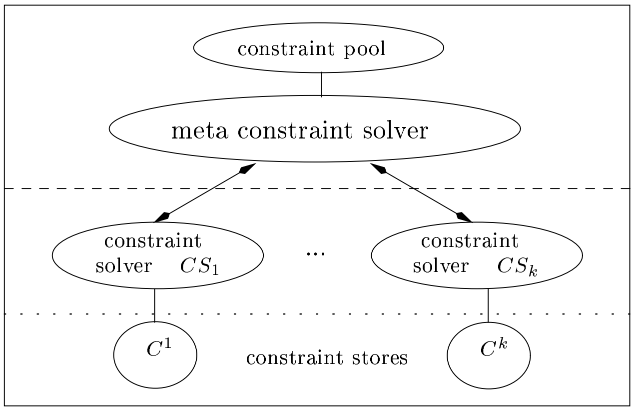


Figure 1. Architecture of the overall system

The meta solver takes constraints from the constraint pool and passes them to the constraint solvers of the corresponding constraint domains (step 1). The individual solvers propagate the received constraints to their stores (step 2). The meta solver forces them to extract information from their stores. This information is added by the meta solver to the constraint pool (step 3). The procedure of steps 1-3 is repeated until the constraint pool contains either the constraint $false$ or the constraint $true$ only. If the pool contains $false$ only, then the initially given conjunction Φ of constraints is unsatisfiable. If it contains $true$ only, then the system could not find a contradiction. Solutions of Φ can be retrieved from the current stores. Because of information exchange between the solvers, each individual solver deals in this way with more information than only that of its associated constraints of Φ.

3.1 A Uniform Interface for Constraint Solvers

To enable a cooperation, the solvers need to exchange information. Let to every constraint system a constraint solver be assigned. Consider a constraint solver CS_ν. Our *uniform interface* of CS_ν consists of a function $tell_\nu$ for *constraint propagation* (according to step 2 of the above behaviour description of the system) and a set of functions $proj_{\nu\to\mu}$ for *constraint projection* (corresponding to the above step 3).

Constraint Propagation The (partial) function $tell_\nu$ is due to constraint satisfaction. $tell_\nu$ adds a constraint $c \in \mathcal{C}ons_\nu$ to a constraint store $C \in \mathcal{DCS}tore_\nu$ if the conjunction of c and C is satisfiable, i.e. if $\mathcal{D} \vDash \exists(C \wedge c)$ holds. Figure 2 shows our requirements to the function $tell_\nu$.

$tell_\nu$: $\mathcal{C}ons_\nu \times \mathcal{DCS}tore_\nu \longrightarrow$
$\{true_{changed}, true_{redundant}, false\} \times \mathcal{DCS}tore_\nu \times \mathcal{DCC}ons_\nu$ with

(1) if $tell_\nu(c, C) = (true_{redundant}, C', C'')$, then
$C' = C$, $C'' = true$, and $\mathcal{D}_\nu \vDash \forall(C \longrightarrow c)$,

(2) if $tell_\nu(c, C) = (true_{changed}, C', C'')$, then
(a) $\mathcal{D}_\nu \vDash \forall((C \wedge c) \longleftrightarrow (C' \wedge C''))$, (b) $\mathcal{D}_\nu \vDash \forall(c \longrightarrow C'')$,
(c) $\mathcal{D}_\nu \vDash \forall(C' \longrightarrow C)$ and (d) $\mathcal{D}_\nu \vDash \exists C'$,

(3) if $tell_\nu(c, C) = (false, C', C'')$, then
$C' = C$, $C'' = false$ and $\mathcal{D}_\nu \nvDash \exists(C \wedge c)$.

Figure 2. Interface function $tell_\nu$ (requirements)

Giving requirements to the interface function $tell_\nu$ instead of a definition enables the integration of a high number of existing solvers into our overall system. The requirements allow to take particular properties of solvers, like their incompleteness or an existing entailment test, into consideration for cost reduction for our overall system. For a detailed description see [3].

Example 1. The interface function $tell_{\mathcal{R}_{lin}}$ of a solver $CS_{\mathcal{R}_{lin}}$ for linear constraints over real numbers could work as follows (an according implementation is possible using the simplex algorithm):

$tell_{\mathcal{R}_{lin}}(c_1, C) = (true_{changed}, C', true)$ with
$$c_1 = (x \leq 3),\ C = true,\ \mathcal{D}_{\mathcal{R}_{lin}} \vDash \forall(C' \longleftrightarrow (c_1 \wedge C)).$$
$tell_{\mathcal{R}_{lin}}(c_2, C') = (true_{redundant}, C', true)$, where $c_2 = (x \leq 4)$ holds.
$tell_{\mathcal{R}_{lin}}(c_3, C') = (false, C', false)$, where $c_3 = (x = 4)$ holds.

Projection of Constraint Stores Constraint projection is used to enable information exchange between constraint solvers. The function $proj_{\nu \to \mu}$ (see Fig.3) projects a store C^ν w.r.t. another constraint system ζ_μ, $\mu \in L \backslash \{\nu\}$ and a set of common variables. It provides knowledge which is implied by the store C^ν of CS_ν in the form of constraints of ζ_μ. The projection function $proj_{\nu \to \mu}$ must be defined in such a way that every solution of C^ν in $\mathcal{D}_\nu$ is a solution of the projection $proj_{\nu \to \mu}(Y, C^\nu)$ in $\mathcal{D}_\mu$, where $Y \subseteq X_\nu \cap X_\mu$. This ensures that projecting a store w.r.t. another constraint system, no solutions of the constraints of the store are lost. We call this required property ***soundness***, its formal description can be found in [3].

$proj_{\nu \to \mu}$: $\mathcal{P}(X_{\nu,\mu}) \times \mathcal{DC}Store_\nu \to \mathcal{DC}Cons_\mu$ with
$$X_{\nu,\mu} = X_\nu \cap X_\mu,\ var(proj_{\nu \to \mu}(Y, C^\nu)) \subseteq Y.$$

Figure 3. Interface function $proj_{\nu \to \mu}$ (requirements)

Example 2. Consider the solver $CS_{\mathcal{R}_{lin}}$ and a solver CS_{FD} of a finite domain constraint system ζ_{FD}. The projection function $proj_{FD \to \mathcal{R}_{lin}}$ of CS_{FD} could be defined on top of a projection function $proj_{FD}$ projecting the store of the finite domain constraint solver CS_{FD} and yielding constraints of $\mathcal{DC}Cons_{FD}$ and a conversion function $conv_{FD \to \mathcal{R}_{lin}} : \mathcal{DC}Cons_{FD} \to \mathcal{DC}Cons_{\mathcal{R}_{lin}}$. The function $proj_{FD \to \mathcal{R}_{lin}}$ could work as follows:
Let $C^{FD} = ((y =_{FD} 3) \wedge (x >_{FD} y) \wedge (x \in_{FD} \{2, 3, 4, 5, 6\}))$ hold.

$$proj_{FD}(\{x\}, C^{FD}) = (x \in_{FD} \{4, 5, 6\}) \text{ and}$$
$$\begin{aligned} proj_{FD \to \mathcal{R}_{lin}}(\{x\}, C^{FD}) &= conv_{FD \to \mathcal{R}_{lin}}(proj_{FD}(\{x\}, C^{FD})) \\ &= ((x \geq 4) \wedge (x \leq 6)). \end{aligned}$$

In the following, we require given computable functions $tell_\nu$ and $proj_{\nu \to \mu}$, $\nu, \mu \in L$.

3.2 Description of the System Behaviour

The behaviour of our system is described by means of reduction relations for ***overall configurations***. An overall configuration $\mathcal{H}$ consists of a *formal disjunction* $\dot{\bigvee}_{i \in \{1,\ldots,m\}} \mathcal{G}_i$ of configurations $\mathcal{G}_i$. Formal disjunction $\dot{\vee}$ is commutative and associative. A *configuration* $\mathcal{G} = (\mathcal{P} \odot \bigwedge_{\nu \in L} C^\nu)$ corresponds to the architecture of the overall system (Fig.1). It consists of the constraint pool $\mathcal{P}$ which is a set of constraints which we want to solve and the conjunction $\bigwedge_{\nu \in L} C^\nu$ of constraint stores. In [2] we show elaborately how to define ***strategies for cooperating constraint solvers***, i.e. reduction systems for overall configurations using the interface functions of the solvers.

In general, in one derivation step one or more configurations $\mathcal{G}_i$, $i \in \{1, \ldots, m\}$, are rewritten by a formal disjunction $\mathcal{HG}_i$ of configurations:

$$OConf1 = \mathcal{H}_1 \dot{\vee} \mathcal{G}_1 \quad \dot{\vee} \ldots \dot{\vee} \mathcal{H}_i \dot{\vee} \mathcal{G}_i \quad \dot{\vee} \ldots \dot{\vee} \mathcal{H}_m \dot{\vee} \mathcal{G}_m \quad \dot{\vee} \mathcal{H}_{m+1} \quad \Longrightarrow$$
$$OConf2 = \mathcal{H}_1 \dot{\vee} \mathcal{H}\mathcal{G}_1 \dot{\vee} \ldots \dot{\vee} \mathcal{H}_i \dot{\vee} \mathcal{H}\mathcal{G}_i \dot{\vee} \ldots \dot{\vee} \mathcal{H}_m \dot{\vee} \mathcal{H}\mathcal{G}_m \dot{\vee} \mathcal{H}_{m+1}$$

Thus, it is useful, to define first a derivation relation for configurations and, based on this, to define a derivation relation for overall configurations.

Using such a *two-step frame* (see [2, 3]) different reduction systems which realize different cooperation strategies for the solvers have be described. The reduction systems allow the derivation of an *initial overall configuration* $\mathcal{G}_0 = \mathcal{P}_\Phi \odot \bigwedge_{\nu \in L} C_0^\nu$, where the constraint pool $\mathcal{P}_\Phi$ contains the constraints of the conjunction Φ which we want to solve and all constraint stores C_0^ν, $\nu \in L$, contain the constraint *true* only. From the derived normal form we obtain information about the satisfiability of the initially given disjunction of constraint conjunctions.

4 Declarative Languages as Solvers

Our system of cooperating solvers allows to integrate different host languages by treating them as constraint solvers. In the following, we consider the integration of a logic language into our system in detail and we sketch the integration of a functional logic language. While we extend the languages by constraints, the evaluation mechanisms of the languages are nearly unchanged, they are only extended by a mechanism for collecting constraints of other constraint systems. The combination of arbitrary constraint systems and languages allows to build constraint languages matching the targeted problems and according to current requirements.

4.1 Logic Programming

A *logic program* P usually consists of a sequence of rules of the form

$$q(s_1, \ldots, s_m) :- q_1(s_{1,1}, \ldots, s_{1,n}), \ldots, q_k(s_{k,1}, \ldots, s_{k,r}).,$$

$k \geq 0$, where every s_i, $s_{i,j}$ are terms and q, q_i are predicate symbols. We also write $Q :- Q_1, \ldots, Q_k.$ in the following. Q_i is called a *literal*. With $Q :- Q_1, \ldots, Q_k.$, where $k = 0$, we denote a rule of the form $Q.$, i.e. a so called *fact*. The aim of the evaluation of a logic program P with a goal $G = (?{-}R_1, \ldots, R_l.)$ $(?{-}R_1, \ldots, R_l.$ stands for $\forall(\neg R_1 \vee \ldots \vee \neg R_l))$ is to find a refutation of G from P (expressed by the empty clause $\Box$) using (SLD-)resolution. If a refutation can be computed, i.e. $P \vDash \exists(R_1 \wedge \ldots \wedge R_l)$ holds, then the computation yields an (so called) *answer substitution* σ such that $P \vDash \forall\, \sigma(R_1 \wedge \ldots \wedge R_l)$ holds. For a detailed description see for example [7].

A *substitution* σ is a function $\sigma : X \to \mathcal{T}(F, X)$ with $\sigma(x) \in \mathcal{T}(F, X)^s$ for every $x \in X^s$ and $dom(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. If $dom(\sigma) = \{x_1, \ldots, x_n\}$ and $\sigma(x_i) = t_i$, we write $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$. The function σ is extended to $\tilde{\sigma} : \mathcal{T}(F, X) \to \mathcal{T}(F, X)$ by $\tilde{\sigma}(x) = \sigma(x)$, for every $x \in X$, and $\tilde{\sigma}(f(t_1, \ldots, t_n)) = f(\tilde{\sigma}(t_1), \ldots, \tilde{\sigma}(t_n))$. In the following, a substitution σ is identified with its extension $\tilde{\sigma}$. The *composition of some substitutions* σ *and* ϕ is defined by $(\sigma \circ \phi)(t) = \sigma(\phi(t))$ for every $t \in \mathcal{T}(F, X)$. A substitution σ is a *unifier* of two terms or literals Q_1 and Q_2 if $\sigma(Q_1) = \sigma(Q_2)$. A unifier σ is a *most general unifier* of Q_1 and Q_2, i.e. $\sigma = mgu(Q_1, Q_2)$, if for every unifier ϕ of Q_1 and Q_2 there exists a substitution ψ such that $\phi = \psi \circ \sigma$.

A substitution σ is *idempotent* iff $\sigma \circ \sigma = \sigma$ holds. The substitutions which we are handling are usually idempotent. Let the *parallel composition* $\uparrow$ *of idempotent substitutions* be defined as given in [8].

4.2 A Logic Language as Constraint Solver

Pure logic programming is convenient for example for working with lists and it allows to handle problems with natural numbers by representing them via constructors, i.e. via the 0-ary constructor `0` and the unary constructor `s`. However, there are problems which cannot be handled comfortably with a pure logic language. Thus, constraints have been integrated which led to constraint logic programming. A *constraint logic program* P consists of a sequence of rules of the form $Q : -Q_1, \ldots, Q_k.$, $k \geq 0$, where every Q_i, $i \in \{1, \ldots, k\}$, may be a constraint of an arbitrary constraint system or a literal. The evaluation mechanism of a constraint logic language handles literals as before and it collects the constraints to check their satisfiability.

Example 3 (logic programs vs. logic programs with constraints). Pure logic languages are not suitably for modelling and solving usual problems which arise when reasoning about electric circuits with resistors. First, because statements like $\mathtt{1/R} = \mathtt{1/R_1} + \mathtt{1/R_2}$ for computing the value of resistors connected in parallel cannot be expressed conveniently. Secondly, because to model real world problems we often want to work with large numbers, for example resistors may have values of some $10^3\Omega$. Using numbers built of constructors may cause much overhead, for example, computing `add(s(s(...s(0)...)),X,Y)` causes a traversal through the whole first argument term `s(s(...s(0)...))` which may have a large depth.

Given three resistors of $10^2\Omega, 2 * 10^2\Omega$ and $10 * 10^2\Omega$, a rule for the sequential composition of resistors, and rules for addition, an associated pure logic program P is the following (at this, we work with resistor values of $10^2\Omega$.):

```
res(simple(s(0)),s(0)).
res(simple(s(s(0))),s(s(0))).
res(simple(s(s(s(s(s(s(s(s(s(s(0})))))))))),
   s(s(s(s(s(s(s(s(s(s(0))))))))))).
res(seq(X,Y),Z) :- res(X,XV),res(Y,YV),add(XV,YV,Z).
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

The integration of constraints of an adequate constraint system $\zeta_{\mathcal{R}}$ for constraints over real numbers allows to formulate constraints with numbers and even the formulation of a rule for the parallel composition of resistors. An associated constraint logic program P' is the following:

$\mathtt{resc}(\mathtt{simple}(10^2), 10^2).$
$\mathtt{resc}(\mathtt{simple}(2 * 10^2), 2 * 10^2).$
$\mathtt{resc}(\mathtt{simple}(10^3), 10^3).$
$\mathtt{resc}(\mathtt{seq}(\mathtt{X}, \mathtt{Y}), \mathtt{Z}) : -\mathtt{resc}(\mathtt{X}, \mathtt{XV}), \mathtt{resc}(\mathtt{Y}, \mathtt{YV}), \mathtt{XV} + \mathtt{YV} =_{\mathcal{R}} \mathtt{Z}.$
$\mathtt{resc}(\mathtt{par}(\mathtt{X}, \mathtt{Y}), \mathtt{Z}) : -\mathtt{resc}(\mathtt{X}, \mathtt{XV}), \mathtt{resc}(\mathtt{Y}, \mathtt{YV}), \mathtt{1/XV} + \mathtt{1/YV} =_{\mathcal{R}} \mathtt{1/Z}.$

Now, we integrate a logic language into our system of cooperating constraint solvers by considering the language together with its evaluation mechanism as constraint solver $CS_{\mathcal{LL}}$ for constraints over the herbrand universe. The constraint system of this solver must contain besides the symbols and predicates introduced by the program the symbols, the predicates, and the functions of all other involved constraint systems. Its constraints are, thus, constraints of $\mathcal{C}ons_\nu$, $\nu \in L$, or literals according to the given program or they are of the form $(\mathtt{X} =_{\mathcal{LL}} \mathtt{t})$, where $\mathtt{X} \in X$ and $\mathtt{t}$ is a term. We define the necessary interface functions $tell_{\mathcal{LL}}$ and $proj_{\mathcal{LL} \to \nu}$, $\nu \in L$, as given in Fig.4

and Fig.5, respectively. At this, for a substitution ϕ, $C(\phi) = \bigwedge_{\mathtt{X} \in dom(\phi)} (\mathtt{X} =_{\mathcal{LC}} \phi(\mathtt{X}))$ denotes a constraint store resp. a constraint conjunction which contains the bindings of a substitution ϕ.

$tell_{\mathcal{LC}}$: Let $C^{\mathcal{LC}} = C(\phi)$ be the current constraint store of $CS_{\mathcal{LC}}$.

Let P be a constraint logic program, let $c = p(t_1, \ldots, t_m) \in \mathcal{C}ons_{\mathcal{LC}}$ be the constraint which is to be propagated.

- Let $\{r_1, \ldots, r_n\} \subseteq P$ be the largest set of rules of P such that for every r_i, $i \in \{1, \ldots, n\}$, there is a variant $r'_i = Q_i : -Q_{i,1}, \ldots, Q_{i,k}$, $k \geq 0$, of r_i s.t. no variable occurs in c and r'_i, and σ_i is a most general unifier of $\phi(c)$ and Q_i, then
$tell_{\mathcal{LC}}(c, C^{\mathcal{LC}}) = (true_{changed}, C'^{\mathcal{LC}},$
$\bigvee_{i \in \{1,\ldots,n\}} (C(\sigma_i) \wedge \sigma_i(\phi(Q_{i,1})) \wedge \ldots \wedge \sigma_i(\phi(Q_{i,k}))))$ holds.
The constraint store does not change, i.e. $C'^{\mathcal{LC}} = C^{\mathcal{LC}}$ holds.
- If there is no rule in P for which the above item holds, then
$tell_{\mathcal{LC}}(c, C^{\mathcal{LC}}) = (false, C^{\mathcal{LC}}, false)$ holds.

Let $c = C(\sigma) \in \mathcal{C}ons_{\mathcal{LC}}$ be the constraint which is to be propagated.

- If $\uparrow (\sigma, \phi) \neq \emptyset$, then $tell_{\mathcal{LC}}(c, C^{\mathcal{LC}}) = (true_{changed}, C'^{\mathcal{LC}}, true)$,
where $C'^{\mathcal{LC}} = C(\uparrow (\sigma, \phi))$.
- If $\uparrow (\sigma, \phi) = \emptyset$, then $tell_{\mathcal{LC}}(c, C^{\mathcal{LC}}) = (false, C^{\mathcal{LC}}, false)$.

Figure 4. Interface function $tell_{\mathcal{LC}}$

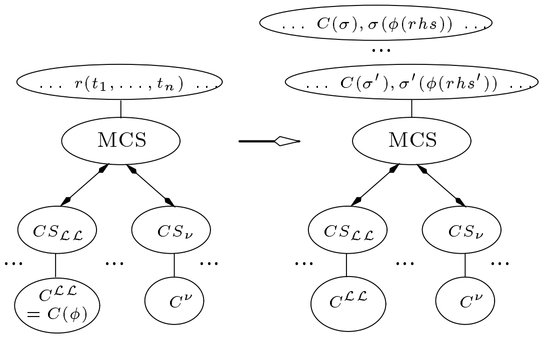
$proj_{\mathcal{LC} \to \nu}$: The projection of a store $C^{\mathcal{LC}} = C(\phi)$ w.r.t. a constraint system $\nu \in L$ and a set of variables $X = X_{\mathcal{LC}} \cap X_\nu$ makes the substitutions for $x \in X$ explicit:

$$proj_{\mathcal{LC} \to \nu}(X, C(\phi)) = \begin{cases} \bigwedge_{x \in X, x \in dom(\phi)} (\phi(x) =^s_\nu x) & \text{if } \exists x \in X \text{ with } \phi(x) \neq x \\ true & \text{otherwise.} \end{cases}$$

Figure 5. Interface function $proj_{\mathcal{LC} \to \nu}, \nu \in L$

Figure 6 illustrates, how the application of $tell_{\mathcal{LC}}$ is used in our framework to simulate a resolution step. First the constraint pool contains the constraint $c = r(t_1, \ldots, r_n)$, the constraint store $C^{\mathcal{LC}}$ contains a substitution ϕ, i.e. $C^{\mathcal{LC}} = C(\phi)$ holds. The successful propagation of c corresponds to a resolution step on c. According to the first item of the $tell_{\mathcal{LC}}$ description, for the constraint $c = r(t_1, \ldots, t_n)$ for every rule $r(s_1, \ldots, s_n) : - \ rhs$ with unifiable left hand side the corresponding most general unifier σ is built. The constraint c in the constraint pool is replaced by the right hand side of the rule under σ and ϕ, i.e. by $\sigma(\phi(rhs))$, and by a constraint conjunction $C(\sigma)$ which expresses the newly built substitution σ. If there is more than one matching rule we get a number of newly built constraint pools and, thus, a number of instantiations of the architecture. This is expressed in our overall framework by an overall configuration consisting of a number of configurations.

The requirements for $tell_{\mathcal{LC}}$ and $proj_{\mathcal{LC} \to \nu}$, $\nu \in L$, according to Fig.2 and Fig.3 are fulfilled, respectively. Notice in particular the first case of the definition of $tell_{\mathcal{LC}}$,



Program:

$r(s_1, \ldots, s_n) :- \underbrace{Q_1, \ldots, Q_m.}_{rhs}$

$r(s'_1, \ldots, s'_n) :- rhs'$

$\ldots$

with

$\sigma = mgu(\phi(r(t_1, \ldots, t_n)),$
$\quad r(s_1, \ldots, s_n))$ and
$\sigma' = mgu(\phi(r(t_1, \ldots, t_n)),$
$\quad r(s'_1, \ldots, s'_n)).$

Figure 6. How the interface function $tell_{\mathcal{LL}}$ works

where

$$\mathcal{P}^\star, \mathcal{D}_\nu, \nu \in L \vDash \forall((C^{\mathcal{LL}} \wedge c) \longleftrightarrow (C'^{\mathcal{LL}} \wedge (\bigvee_{i \in \{1,\ldots,n\}} (C(\sigma_i) \wedge \sigma_i(\phi(Q_{i,1})) \wedge \ldots \wedge \sigma_i(\phi(Q_{i,k}))))))$$

holds (see for example [5] and there in particular Lemma 5.3).

In Example 4 we demonstrate the use of the logic language as constraint solver by means of a part of a derivation of an initial overall configuration using our defined interface functions.

Example 4. Given the constraint logic program P' of Example 3 using our system of cooperating constraint solvers with a solver for arithmetic constraints over real numbers $CS_{\mathcal{R}}$ and the solver $CS_{\mathcal{LL}}$ based on the logic language together with P', we get the following trace (we omit unnecessary parts of substitutions to shorten the trace):

$\{\texttt{resc(par(R1, seq(R2, R3)), 75)}\} \odot C_0^{\mathcal{LL}} \wedge C_0^{\mathcal{R}} \Longrightarrow$
by $tell(\texttt{resc(par(R1, seq(R2, R3)), 75)}, C_0^{\mathcal{LL}}) = (true_{changed}, C_0^{\mathcal{LL}},$
$\texttt{resc(R1, RV1)} \wedge \texttt{resc(seq(R2, R3), RV23)} \wedge \texttt{1/RV1} + \texttt{1/RV23} =_{\mathcal{R}} \texttt{1/75})$
$\{\texttt{resc(R1, RV1)}, \texttt{resc(seq(R2, R3), RV23)}, \texttt{1/RV1} + \texttt{1/RV23} =_{\mathcal{R}} \texttt{1/75}\}$
$\odot C_0^{\mathcal{LL}} \wedge C_0^{\mathcal{R}} \Longrightarrow$
by $tell(\texttt{resc(R1, RV1)}, C_0^{\mathcal{LL}}) = (true_{changed}, C_0^{\mathcal{LL}},$
$(\texttt{R1} =_{\mathcal{LL}} \texttt{simple}(10^2) \wedge \texttt{RV1} =_{\mathcal{LL}} 10^2) \vee \ldots)$

Obviously, there are five alternatives according to P'. In the following, we only derive the first alternative and leave the others out (which is marked by $\ldots$).

$(\{\texttt{R1} =_{\mathcal{LL}} \texttt{simple}(10^2), \texttt{RV1} =_{\mathcal{LL}} 10^2, \texttt{resc(seq(R2, R3), RV23)},$
$\texttt{1/RV1} + \texttt{1/RV23} =_{\mathcal{R}} \texttt{1/75}\} \odot C_0^{\mathcal{LL}} \wedge C_0^{\mathcal{R}}) \dot{\vee} \ldots \Longrightarrow$
by $tell(\texttt{R1} =_{\mathcal{LL}} \texttt{simple}(10^2), C_0^{\mathcal{LL}}) = (true_{changed}, C_1^{\mathcal{LL}}, \texttt{true})$ and
$tell(\texttt{RV1} =_{\mathcal{LL}} 10^2, C_1^{\mathcal{LL}}) = (true_{changed}, C_2^{\mathcal{LL}}, \texttt{true})$, where
$C_2^{\mathcal{LL}} = C(\{\texttt{R1/simple}(10^2), \texttt{RV1/}10^2\})$
$(\{\texttt{resc(seq(R2, R3), RV23)}, \texttt{1/RV1} + \texttt{1/RV23} =_{\mathcal{R}} \texttt{1/75}\} \odot C_2^{\mathcal{LL}} \wedge C_0^{\mathcal{R}})\ \dot{\vee} \ldots$

This first configuration of our overall configuration corresponds to the situation in the computation of a constraint logic program, where a resolution step for $\texttt{resc(par(R1,seq(R2,R3)),75)}$ has been performed and as well a following resolution step for the newly received goal $\texttt{resc(R1,RV1)}$ with the first rule with unifying left hand side. The pool describes the goal which is now to be solved and the constraint store $C_2^{\mathcal{LL}}$ contains the computed substitution. To distribute this substitution over the full goal we project $C_2^{\mathcal{LL}}$ w.r.t. $\zeta_{\mathcal{R}}$ such that $CS_{\mathcal{R}}$ gets the information about the bindings too:

$(\{\texttt{resc(seq(R2,R3),RV23)}, \texttt{1/RV1} + \texttt{1/RV23} =_{\mathcal{R}} \texttt{1/75}\} \odot C_2^{\mathcal{LL}} \wedge C_0^{\mathcal{R}})\ \dot{\vee} \ldots \Longrightarrow$

$$\begin{aligned} \text{by } proj_{\mathcal{LL}\to\mathcal{R}}(\{\texttt{RV1}\}, C_2^{\mathcal{LL}}) &= conv_{\mathcal{LL}\to\mathcal{R}}(proj_{\mathcal{LL}}(\{\texttt{RV1}\}, C_2^{\mathcal{LL}})) \\ &= conv_{\mathcal{LL}\to\mathcal{R}}(\texttt{RV1} =_{\mathcal{LL}} \texttt{10}^2) = (\texttt{RV1} =_{\mathcal{R}} \texttt{10}^2) \text{ and} \\ proj_{\mathcal{LL}\to\mathcal{R}}(\{\texttt{R1}\}, C_2^{\mathcal{LL}}) &= conv_{\mathcal{LL}\to\mathcal{R}}(proj_{\mathcal{LL}}(\{\texttt{R1}\}, C_2^{\mathcal{LL}})) \\ &= conv_{\mathcal{LL}\to\mathcal{R}}(\texttt{R1} =_{\mathcal{LL}} \texttt{simple}(\texttt{10}^2)) = \texttt{true}^1 \end{aligned}$$

$$(\{\texttt{RV1} =_{\mathcal{R}} \texttt{10}^2, \texttt{true}, \texttt{resc(seq(R2,R3),RV23)}, \texttt{1/RV1} + \texttt{1/RV23} =_{\mathcal{R}} \texttt{1/75}\} \odot C_2^{\mathcal{LL}} \wedge C_0^{\mathcal{R}})\ \dot{\vee} \ldots$$

The propagation of all constraints of the pool, the projection of the computed bindings, and propagating the newly received projections again yields the following overall configuration:

$(\{\texttt{true}\} \odot C^{\mathcal{LL}} \wedge C^{\mathcal{R}})\dot{\vee} \ldots$

Projecting $C^{\mathcal{LL}}$ w.r.t. the variables of the initial constraint $\texttt{resc(par(R1,seq(R2,R3)),75)}$ yields the valid bindings:

$proj_{\mathcal{LL}}(\{\texttt{Ri}\}, C^{\mathcal{LL}}) = (\texttt{Ri} =_{\mathcal{LL}} \texttt{simple}(\texttt{10}^2))$, $\texttt{i} \in \{1,2\}$, and

$proj_{\mathcal{LL}}(\{\texttt{R3}\}, C^{\mathcal{LL}}) = (\texttt{R3} =_{\mathcal{LL}} \texttt{simple}(\texttt{2} * \texttt{10}^2))$ holds.

The computed bindings for R1, R2, and R3 are as expectedly,

$\texttt{resc(par(simple(}\texttt{10}^2\texttt{),seq(simple(}\texttt{10}^2\texttt{),simple(}\texttt{2}*\texttt{10}^2\texttt{))),75)}$ holds.

The correspondence between a resolution sequence and a derivation using our system of cooperating solvers can be observed. However, using our system, different configurations of an overall configuration allow to consider all possible resolution sequences.

4.3 Functional Logic Programs

Now, let us have a short look at a second language: a functional logic one. In contrast to logic languages which work with predicates functional logic ones allow to work with functions which may be even arguments of functions. Let $\Sigma = (S, F, R; ar)$ be a signature, where F is partitioned into a set Δ of *constructors* and a set Γ of *defined functions.* A *functional logic program* P over Σ is a finite set of rules of the form $f(t_1, \ldots, t_n) \to r$, where $f \in \Gamma^{s_1 \times \ldots \times s_n \to s}$, $t_i \in \mathcal{T}(\Delta, X)^{s_i}$, and $r \in \mathcal{T}(F, X)^s$. $f(t_1, \ldots, t_n)$ is *linear,* i.e. it does not contain multiple occurrences of one variable, and $var(r) \subseteq var(f(t_1, \ldots, t_n))$ holds. In the usual way, P induces a congruence relation $=_P$. A typical evaluation mechanism for functional logic programs is narrowing [1].

4.4 A Functional Logic Language as Constraint Solver

The introduction of constraints into the rules of our language yields constraint functional logic programming. A *functional logic program P with constraints over Σ* is a

[1] Since $\texttt{simple} \notin F_{\mathcal{R}}$ holds, this projection is 'translated' to $\texttt{true}$.

finite set of rules of the form $(f(t_1,\ldots,t_n) \to r \texttt{ where } G)$, where $f \in \Gamma^{s_1\times\ldots\times s_n\to s}$, $t_i \in \mathcal{T}(\Delta,X)^{s_i}$, and $r \in \mathcal{T}(F,X)^s$. $f(t_1,\ldots,t_n)$ is linear, $var(r) \subseteq var(f(t_1,\ldots,t_n))$ holds, and G is a finite set of constraints over Σ.

To handle the constraints during the evaluation of a functional logic program we need to extend narrowing by constraints. A *position* p in a term t is represented by a sequence of natural numbers, $t|_p$ denotes the *subterm* of t at position p, and $t[r]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term r. A *narrowing step with constraints* $t \rightsquigarrow_{s,p,\sigma} (t',C')$, where t,t' are terms and C' is a set of constraints, is defined as follows: t is *narrowable* to (t',C') if there is a nonvariable position p in t, i.e. $t|_p \notin X$, $s = (l \to r \texttt{ where } G)$ is a new variant of a rule from P, $\sigma = mgu(t|_p, l)$, and $t' = \sigma(t[r]_p)$ and $C' = \bigwedge_{c\in G} \sigma(c)$.

Now, we consider a functional logic language together with its evaluation mechanism as constraint solver $CS_{\mathcal{FL}}$ for constraints over functional expressions. As for the logic language, the constraint system $\zeta_{\mathcal{FL}}$ must contain the symbols, the predicates, and the functions of all other involved constraint systems as well. Constraints of $\zeta_{\mathcal{FL}}$ are restricted to be of the form $(\texttt{t} =_P \texttt{X})$, where $\texttt{t} \in \mathcal{T}(F,X)$ and $\texttt{X} \in X$. Constraints of the form $\texttt{t}_1 =_P \texttt{t}_2$, where $\texttt{t}_1, \texttt{t}_2 \in \mathcal{T}(F,X)$, are decomposed into $(\texttt{t}_1 =_P \texttt{X} \wedge \texttt{t}_2 =_P \texttt{X})$. Let for a substitution ϕ, $C(\phi) = \bigwedge_{\texttt{X}\in dom(\phi)} (\texttt{X} =_{\mathcal{FL}} \phi(\texttt{X}))$ hold.

Instead of giving a formal definition we only sketch what 'propagating constraints' means in this context and how narrowing steps with constraints are used there.

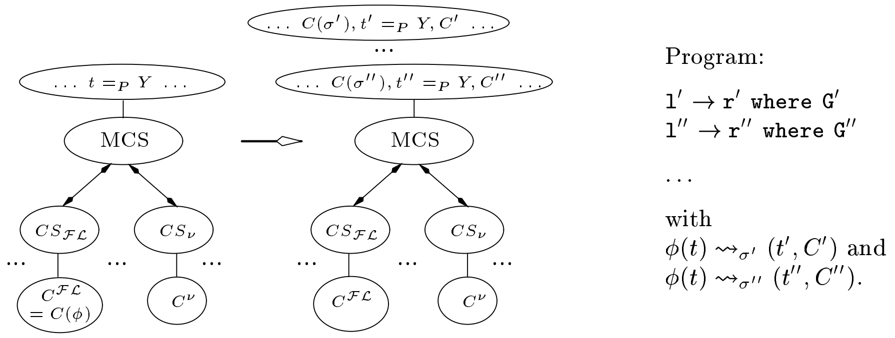


Figure 7. How the interface function $tell_{\mathcal{FL}}$ works

Consider Fig.7 which illustrates, how the application of $tell_{\mathcal{FL}}$ works in our framework to simulate a narrowing step with constraints. Initially the constraint pool contains the constraint $c = (t =_P Y)$ and the constraint store $C^{\mathcal{FL}}$ contains a substitution ϕ. The successful propagation of c corresponds to a narrowing step with constraints on t. For the term t for every matching rule $\texttt{l} \to \texttt{r where G}$ a narrowing step with constraints is performed yielding the most general unifier σ' and a tuple (t',C'). The constraint $(t =_P Y)$ in the constraint pool is replaced by the the constraint $(t' =_P Y)$, by a constraint conjunction $C(\sigma')$ which expresses the newly built substitution σ', and by the newly built constraint conjunction C' which is arisen from the narrowing step. If there is more than one matching rule we get a number of newly built constraint pools and, thus, a number of instantiations of the architecture. This would be expressed in our overall framework by an overall configuration consisting of a number of configurations.

Figure 7 illustrates a successful constraint propagation, the cases of

1. a failing propagation because there is no matching rule for the term to be reduced – this yields a constraint $false$ which is added to the pool and
2. a propagation, where the term t is already a constructor term, – a binding of Y to this term is tried to add to the constraint store $CS^{\mathcal{FL}}$ by parallel composition of substitutions [8]

are left out here.

The definition of $proj_{\mathcal{FL}\to\nu}, \nu \in L$, is the same as that of $proj_{\mathcal{LL}\to\nu}$ in Fig.5 (where every index $\mathcal{LL}$ is replaced by $\mathcal{FL}$).

5 Conclusion

This paper shortly describes a general approach for the integration of arbitrary declarative languages and constraint systems. After a short reintroduction of a system of cooperating solvers of [2, 3] we have shown how to integrate host languages, in particular a logic and a functional logic one, into such a system by treating the evaluation mechanisms of the languages together with programs as constraint solvers and defining interface functions for them.

In contrast to several other systems and schemes for the combination of constraint solvers [4, 6, 9] which usually have one fixed host language (a logic one), our system allows to integrate different host languages. Moreover, our system is very flexible, because we can integrate different solvers and we can define different cooperation strategies (shown in [2]). Thus, our approach allows to build constraint languages according to current requirements and, thus, comfortable modelling and solving of a wide range of problems.

References

1. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
2. P. Hofstedt. Better Communication for Tighter Cooperation. In *First International Conference on Computational Logic*, volume 1861 of *LNCS*. Springer, 2000.
3. P. Hofstedt. *Cooperation and Coordination of Constraint Solvers*. PhD thesis, Dresden University of Technology, 2001.
4. H. Hong. Confluency of Cooperative Constraint Solvers. Technical Report 94-08, Research Institute for Symbolic Computation, Linz, Austria, 1994.
5. J. Jaffar, M.J. Maher, K. Marriott, and P. Stuckey. The Semantics of Constraint Logic Programs. *Journal of Logic Programming*, 37:1–46, 1998.
6. E. Monfroy. *Solver Collaboration for Constraint Logic Programming*. PhD thesis, Centre de Recherche en Informatique de Nancy. INRIA-Lorraine, 1996.
7. U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog*. John Wiley & Sons Ltd., 1995.
8. C. Palamidessi. Algebraic Properties of Idempotent Substitutions. In M.S. Paterson, editor, *Automata, Languages and Programming - ICALP*, volume 443 of *LNCS*. Springer, 1990.
9. M. Rueher. An Architecture for Cooperating Constraint Solvers on Reals. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer, 1995.

An Implementation of Narrowing Strategies*

Sergio Antoy[1], Michael Hanus[2], Bart Massey[1], and Frank Steiner[2]

[1] Department of Computer Science, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A.
{antoy,bart}@cs.pdx.edu

[2] Institut für Informatik, Christian-Albrechts-Universität Kiel,
Olshausenstr. 40, D-24098 Kiel, Germany
{mh,fst}@informatik.uni-kiel.de

Abstract This paper describes an implementation of narrowing, an essential component of implementations of modern functional logic languages. These implementations rely on narrowing, in particular on some optimal narrowing strategies, to execute functional logic programs. We translate functional logic programs into imperative (Java) programs without an intermediate abstract machine. A central idea of our approach is the explicit representation and processing of narrowing computations as data objects. This enables the implementation of operationally complete strategies (i.e., without backtracking) or techniques for search control (e.g., encapsulated search). Thanks to the use of an intermediate and portable representation of programs, our implementation is general enough to be used as a common back end for a wide variety of functional logic languages.

1 Introduction

This paper describes an implementation of narrowing for overlapping inductively sequential rewrite systems [5]. Narrowing is the essential computational engine of functional logic languages (see [13] for a survey on such languages and their implementations). An implementation of narrowing translates a program consisting of rewrite rules into executable code. This executable code currently falls into two categories: Prolog predicates (e.g., [4, 11, 14, 25]) or instructions for an abstract machine (e.g., [10, 18, 24, 27]). Although these approaches are relatively simple, in both cases, several layers of interpretation separate the functional logic program from the hardware intended to execute it. Obviously, this situation does not lead to efficient execution.

In this paper we investigate a different approach. We translate a functional logic program into an imperative program. Our target language is Java, but we make limited use of specific object-oriented features, such as inheritance and dynamic polymorphism. Replacing Java with a lower-level target language, such as C or machine code, would be a simple task.

In Section 2 we briefly introduce the aspects of functional logic programming relevant to our discussion. In Section 3 we describe the elements and the characteristics

* This research has been partially supported by the DAAD/NSF under grant INT-9981317 and the German Research Council (DFG) under grant Ha 2457/1-2. This paper is an abridgement of a paper to appear in the proceedings of the *Third International Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, and is copyright 2001 by the Association for Computing Machinery. Extracts of that paper are reproduced for this purpose by permission of the ACM.

of our implementation of narrowing. In Section 4 we describe aspects of our compilation process, as well as execution issues such as input, output and tracing/debugging that may greatly affect the usability of a system. In Section 5 we summarize current efforts toward the implementation of functional logic languages, particularly w.r.t. implementations of narrowing and how they compare to our work. Section 6 offers some conclusions.

2 Functional Logic Programs

Functional logic languages combine the operational principles of two of the most important declarative programming paradigms, namely functional and logic programming (see [13] for a survey). Efficient demand-driven functional computations are amalgamated with the flexible use of logical variables, providing for function inversion and search for solutions. Functional logic languages with a sound and complete operational semantics are usually based on narrowing (originally introduced in automated theorem proving [29]) which combines reduction (from the functional part) and variable instantiation (from the logic part). A *narrowing step* instantiates variables of an expression and applies a reduction step to a redex of the instantiated expression. The instantiation of variables is usually computed by unifying a subterm of the entire expression with the left-hand side of some program equation.

Example 1. Consider the following rules defining the $\leq$ predicate **leq** on natural numbers which are represented by terms built from **zero** and **succ**:

```
leq(zero,Y)           = true
leq(succ(X),zero)     = false
leq(succ(X),succ(Y))  = leq(X,Y)
```

The expression **leq(succ(M),Y)** can be evaluated (i.e., reduced to a value) by instantiating **Y** to **succ(N)** to apply the third equation, followed by the instantiation of **M** to **zero** to apply the first equation:

$$\texttt{leq(succ(M),Y)} \leadsto_{\{\texttt{Y}\mapsto\texttt{succ(N)}\}} \texttt{leq(M,N)} \leadsto_{\{\texttt{M}\mapsto\texttt{zero}\}} \texttt{true}$$

Narrowing provides completeness in the sense of logic programming (computation of all answers, i.e., substitutions leading to successful evaluations) as well as functional programming (computation of values). Since simple narrowing can have a huge search space, a lot of effort has been made to develop sophisticated narrowing strategies without losing completeness (see [13]). *Needed narrowing* [7] is based on the idea of evaluating only subterms which are *needed* in order to compute a result. For instance, in a term like **leq(**t_1**,**t_2**)**, it is always necessary to evaluate t_1 (to some variable or constructor-rooted term) since all three rules in Example 1 have a non-variable first argument. On the other hand, the evaluation of t_2 is only needed if t_1 is of the form **succ(**t**)**. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor term, here **zero** or **succ(V)**. Depending on this instantiation, either the first equation is applied or the second argument t_2 is evaluated. Needed narrowing is currently the best narrowing strategy for first-order (inductively sequential) functional logic programs [3] due to its optimality properties w.r.t. the length of derivations and the independence of computed solutions, and due to the possibility of efficiently implementing needed narrowing by pattern matching and unification [7]. Moreover, it has been extended in various directions, e.g., higher-order functions and λ-terms as data structures [17], overlapping rules [5], and concurrent computations [15].

Needed narrowing is complete, in the sense that for each solution to a goal there exists a narrowing derivation computing a more general solution. However, most of the existing implementations of narrowing lack this property since they are based on Prolog-style backtracking. Since backtracking is not fair in exploring all derivation paths, some solutions might not be found in the presence of infinite derivations, i.e., these implementations are incomplete from an operational point of view. An important property of our implementation is its operational completeness, i.e., all computable answers are eventually computed by our implementation.

3 Implementation of Needed Narrowing

In this section we describe the main ideas of our implementation of narrowing. We implement a strategy, referred to as *INS* [5], proven sound and complete for the class of the overlapping inductively sequential rewrite systems. In these systems, the left-hand sides of the rewrite rules defining an operation can be organized in definitional trees. However, an operation may have distinct rewrite rules with the same left-hand side (modulo renaming of variables): operation **`coin`** (Section 3.8), is one example. To ease the understanding of our work, we first describe the implementation of rewrite computations in inductively sequential rewrite systems. We then describe the extensions that lead to narrowing in overlapping inductively sequential rewrite systems.

3.1 Overview

The overall goals of our implementation are speed of execution and operational completeness. The following principles guide our implementation and are instrumental in achieving the goal.

1. A reduction step replaces a redex of a term with its reduct. A term is represented as a tree-like data structure. The execution of a reduction updates only the portion of this data structure affected by the replacement. Thus, the cost of a reduction is independent of its context. We call this principle *in-place* replacement.
2. Only somewhat needed steps are executed. We use the qualifier "somewhat" because different notions of *need* have been proposed for different classes of rewrite systems. We execute a particular kind of steps that for reductions in orthogonal systems is known as *root-needed* [28]. Thus, reductions that are a priori useless are never performed. We call this principle *useful step.*
3. *Don't know* non-deterministic reductions are executed in parallel. Both narrowing computations (in most rewrite systems) and reductions (in interesting rewrite systems) are non-deterministic. Without some form of parallel execution, operational completeness would be lost. We call this principle *operational completeness.*

In inductively sequential rewrite systems, and when computations are restricted to rewriting, it is relatively easy to faithfully implement all the above principles. In fact, our implementation does it. However, our environment is considerably richer. We execute *narrowing* computations in *overlapping* inductively sequential rewrite systems. In this situation, two complications arise. The non-determinism of narrowing and/or of overlapping rules imply that a redex may have several replacements. In these situations, there cannot be a single in-place replacement. Furthermore, the steps that we compute in *overlapping* inductively sequential rewrite systems are needed, but only modulo non-deterministic choices [5]. Hence, some step may not be needed in the strict sense of [7,22], but we may not be able to know by feasible means which steps.

The architecture of our implementation is characterized by *terms* and *computations*. Both terms and computations are organized into tree-like linked (dynamic) structures. A *term* consists of a *root symbol* applied to zero or more *arguments* which are themselves terms. A *computation* consists of a stack of *terms* that identify reduction steps. All the terms in the stack, with the possible exception of the top, are not yet redexes, but will eventually become redexes, and be reduced, before the computation is complete. In terms, links go from a parent to its children, whereas in computations links go from children to their parent.

A graphical representation of these objects is shown in Figure 1. In this figure, the steps to the left represent the terms in the stack of the computation. $Step_0$ is the bottom of the stack: it cannot be executed before $Step_1$ is executed. Likewise $Step_1$ cannot be executed before $Step_2$ is executed.

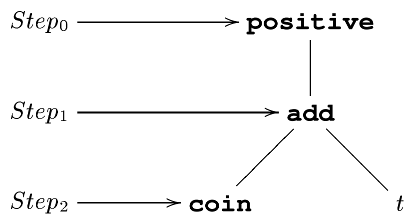


Figure 1. Snapshot of a computation of term **`positive(add(coin,`**t**`))`**

To ease understanding, we begin with an account of our implementation of rewriting computations in inductively sequential rewrite systems. Although non-trivial, this implementation is simple enough to inspire confidence in both its correctness and efficiency. Then, we generalize the discussion to larger classes of rewrite systems and finally to narrowing computations and argue why both correctness and efficiency of this initial implementation are preserved by these extensions.

3.2 Symbol representation

Symbols are used to represent terms. A *symbol* is an object that contains two pieces of information: a *name* and a *kind*. Since there is no good reason to have more than one instance of a given symbol in a program, each distinct symbol is implemented as an immutable singleton object. The *name* is a string. The *kind* is a tag that classifies a symbol. For now, the tag is either "defined operation" or "data constructor". Additional tags will be defined later to compute with larger classes of rewrite systems. The tag of a symbol is used to dispatch computations that depend on the classification of a symbol. Of course, we could dispatch these computations by dynamic polymorphism, i.e., by defining an abstract method overridden by subclasses. Often, these methods would consist of a few statements that use the environment of the caller. A tag avoids both a proliferation of small methods and the inefficiency of passing around the environment. Furthermore, this architecture supports implementations in objectless target languages as well.

Nevertheless, in our Java architecture, class *symbol* has subclasses such as *operation* and *constructor*. In particular, there is one subclass of *operation* for each defined operation f of a functional logic program. This class, according to our second principle, contains the code for the execution of a useful step of any term rooted by f. Operations are defined by rewrite rules. We use the following rules in the examples to come.

```
add (zero, Y)     = Y
add (succ (X), Y) = succ (add (X, Y))

positive (zero)     = false
positive (succ (_)) = true
```

3.3 Term representation

Terms of user-defined type contain two pieces of information: the *root* of the *term*, which is a *symbol*, and the *arguments* of the *root*, which are *terms* themselves. Terms of builtin types contain specialized information, e.g., terms of the builtin type *int* contain an *int*. This situation suggests defining a common base class and a specialization of this class for each appropriate type of term. However, this is in conflict with the fact that according to the first principle of our implementation, a *term* is a mutable object. In Java, the class of an object cannot change during execution.

Therefore, we implement a *term* as a bridge pattern. A term delegates its functionality to a representation. Different types, such as user-defined types, builtin types, and variables are represented differently. All the representations provide a common functionality. The representation of a term object can change at run-time and thus provide mutability of both value and behavior as required by the implementation.

3.4 Computation representation

A *computation* is an object abstracting the necessity to execute a sequence of specific reduction steps in a term. Class *computation* contains two pieces of information:

1. A *stack of terms* to be contracted (reduced at the root). The terms in the stack are not redexes except, possibly, the top term. Each term in the stack is a subterm of the term below it, and must be reduced to a constructor-rooted term in order to reduce the term below it. Therefore, the elements of the stack in a computation may be regarded as steps as well. The underpinning theoretical justification of this stack of steps is in the proof of Th. 24 of the extended version of [5]. We ensure that every term in the stack eventually will be contracted. To achieve this aim, if a complete strategy cannot execute a step in an operation-rooted term, it reduces the term to the special value *failure*.
2. A set of *bookkeeping information*. For example, this information includes the number of steps executed by the computation and the elapsed time. An interesting bookkeeping datum is the state of a computation. Computations being

executed are in a *ready* state. A computation's state becomes *exhausted* after the computation has been executed and it has been determined that no more steps will be executed at the root of the bottom-most term of the stack. Before becoming exhausted a computation state may be either *result* or *failure*. Later, we will extend our model of computation with residuation. With the introduction of residuation, a new state of a computation, *flounder*, is introduced as well.

Loosely speaking, an initial computation is created for an initial top-level expression to evaluate. This expression is the top and only term of the stack of this computation. If the top term t is not a redex, a subterm of t needed to contract t is placed on the stack and so on until a redex is found. A redex on top of the stack is replaced by its reduct. If the reduct is constructor-rooted, the stack is popped (its top element is discarded).

3.5 Search space representation

The search space is a queue of computations which are repeatedly selected for processing. The machinery of a queue and fair selection is not necessary for rewriting in inductively sequential rewrite systems. For these systems, computations are strictly sequential and consequently a single (possibly implicit) stack of steps would suffice. However, the architecture that we describe not only accommodates the extensions from rewriting to narrowing and/or from inductively sequential rewrite systems to the larger classes that are coming later, but it allows us to compute more efficiently.

A computation serves two purposes: (1) finding maximal operation-rooted subterms t of the top-level term to evaluate and (2) reducing each t to head normal form. The pseudo-code of Figure 2 sketches part (2), which is the most challenging. Some optimizations would be possible, but we avoid them for the sake of clarity.

Since inductively sequential rewrite systems are confluent, replacing in-place a subterm u of a term t with u's reduct does not prevent reaching t's normal form. When a term has a result this result is found, since repeated contractions of needed redexes are normalizing.

3.6 Sentinel

The first extension to the previous model is the introduction of a "sentinel" at the root of the top-level expression being evaluated. For this, we introduce a distinguished symbol called *sentinel* that takes exactly one argument of any kind. If t is the term to evaluate, our implementation evaluates $\mathit{sentinel}(t)$ instead. Thus, this is the actual term of the initial computation. Symbol *sentinel* has characteristics of both an operation and a constructor. Similar to an operation, the stack of the initial computation contains $\mathit{sentinel}(t)$, but similar to a constructor, $\mathit{sentinel}(t)$ cannot be contracted for any t. Having a sentinel has several advantages. The strategy works with the sentinel by means of implicit rewrite rules that always look for an internal needed redex and never contract the *sentinel*-rooted term itself. Also, using a sentinel saves frequent tests similar to using a sentinel in many classic algorithms, e.g., sorting.

```
while the queue is not empty
| select a ready computation k from the queue
| let t be the term at the top of k's stack
| switch on the root of t
| | case t is operation-rooted
| | | switch on the reducibility of t
| | | | case t is a redex
| | | | | replace t with its reduct
| | | | | put k back into the queue
| | | | case t is not a redex
| | | | | switch on s, a maximal needed subterm of t
| | | | | | case s exists
| | | | | | | push s on k's stack
| | | | | | | put k back into the queue
| | | | | | case s does not exist
| | | | | | | stop the computation, no result exists
| | | | | endswitch
| | | endswitch
| | case t is constructor-rooted
| | | pop k's stack
| | | if k's stack is not empty
| | | | put k back into the queue
| endswitch
endwhile
```

Figure 2. Procedure to evaluate a term to a head normal form

3.7 Failure

The second extension to the previous model is concerned with the possibility of a "failure" of a computation. A failure occurs when a term has no constructor normal form. The computation detects a failure when the strategy, which is complete, finds no useful steps (redexes) in an operation-rooted term.

The pseudo-code presented earlier simply terminates the computation when it detects a failure. For the extensions discussed later it is more convenient to explicitly represent failures in a term. This allows us, e.g., to clean up computations that cannot be completed and to avoid duplicating certain computations. To this purpose we introduce a new symbol called *failure*. The *failure* symbol is treated as a constant constructor.

Suppose that u is an operation-rooted term. If the strategy finds no step in u, it evaluates u to *failure*. A *failure* symbol is treated as a constructor during the pattern matching process. Implicit rewrite rules for each defined operation rewrite any term t to *failure* when a *failure* occurs at a needed position of t. For example, we perform the following reduction:

add (failure, v**)** $\rightarrow$ **failure**

With these implicit rewrite rules, an inner occurrence of *failure* in a term propagates up to the sentinel, which can thus report that a computation has no result. The explicit representation of failing computations is also important in performing non-deterministic computations.

3.8 Non-determinism

The third extension to the previous model is concerned with non-determinism. In our work, non-determinism is expressed by rewrite rules with identical left-hand sides, but distinct right-hand sides. A textbook example of a non-deterministic defined operation is:

```
coin = zero
coin = succ (zero)
```

This operation differs from the previous ones in that a given term, say $s =$ **coin**, has two distinct reducts.

The most immediate problem posed by non-deterministic operations is that if s occurs in some term t and we replace in-place s with one of its replacements, we may lose a result that could be obtained with another replacement. If a term such as s becomes the top of the stack of a computation k, we change the state of k to *exhausted* and we start two or more new computations. Each new computation, say k', begins with a stack containing a single term obtained by one of the several possible reductions of s.

The procedure described above can be optimized in many ways. We mention only the most important one that we have implemented — the sharing of subterms disjoint from s. We show this optimization in an example. Suppose that the top-level term being evaluated is:

```
add (coin, t)
```

The non-determinism of **coin** gives rise to the computation of the following two terms:

```
add (zero, t)
add (succ (zero), t)
```

These terms are evaluated concurrently and independently. However, term t in the above display is shared rather than duplicated. Sharing improves the efficiency of computations since only one term, rather than several equal copies, is constructed and possibly evaluated. In some situations, a shared term may occur in the stacks of two independent computations and be concurrently evaluated by each computation. This approach avoids a common problem of backtracking-based implementations of functional logic languages, in which t will be evaluated twice if it is needed during the evaluation of both **add** terms shown above.

3.9 Rewrite rules

The final relevant portion of our architecture is the implementation of rewrite rules. All the rules of an ordinary defined operation f are translated into a single Java method. This method implicitly uses a definitional tree of f to compare constructor symbols in inductive positions of the tree with corresponding occurrences in an f-rooted term t to reduce. Let k_t be a computation in the queue, *ready* the state of k_t, and t the term on the top of k_t's stack. The following case breakdown defines the code that needs to be generated.

1. If t is a redex with a single reduct, then t is replaced in-place by its reduct.
2. If t is a redex with several reducts, then a new computation is started for each reduct. The state of k_t is changed to *exhausted*.

3. If in a needed position of t there is *failure*, then t is considered a redex as well and it is replaced in-place by *failure*.
4. If in a needed position of t there is an operation-rooted ordinary term s, then s is pushed on the stack of k_t.
5. The last case to consider is when operation f is incompletely defined and no needed subterm is found in t. In this case, t is replaced in-place by *failure*.

3.10 Narrowing

At this point we are ready to discuss the extension of our implementation to narrowing. A narrowing step instantiates variables in a way very similar to a non-deterministic reduction step. For example, suppose that *allnat* is an operation defined by the rules:

```
allnat = zero
allnat = succ (allnat)
```

Narrowing term **add(X,** t**)**, where **X** is an uninstantiated variable and t is any term, is not much different from reducing **add(allnat,** t**)**.

There are two key differences in the handling of variables w.r.t. non-deterministic reductions: (1) we must keep track of variable bindings to construct the *computed answer* at the end of a computation, and (2) if a given variable occurs repeatedly in a term being evaluated, the replacement of a variable with its binding must replace all the occurrences. We solve point (1) by storing the binding of a variable in a computation. Point (2) is simply bookkeeping. We represent substitutions "incrementally." A computation computes both a value (for the functional part) and an answer (for the logic part). The answer is a substitution. In most cases, a narrowing step produces several distinct bindings for a variable. Each of these bindings increments a previously computed substitution. For example, suppose that the expression to narrow is:

add (X, Y) = t

for some term t. Some computation may initially bind **X** to **zero**. Later on, a narrowing step may bind **Y** independently to both **zero** and **succ(**$\mathbf{Y_1}$**)**. These bindings will "add" to the previous one. The previous binding is shared, which saves both memory and execution time.

3.11 Parallelism

Our implementation includes a form of parallelism known as *parallel-and*. And-parallel steps do not affect the soundness or completeness of the strategy, *INS*, underlying our implementation, but in some cases they may significantly reduce the size of the narrowing space of a computation — possibly from infinite to finite. The *parallel-and* operation is handled explicitly by our implementation. If a computation k leads to the evaluation of t **&** u, where t and u are terms and "**&**" denotes the parallel-and operation, then steps of both t and u are scheduled. This requires to change the stack of a computation into a tree-like structure. The set of leaves of this tree-like structure replaces the top of the stack previously discussed.

As soon as one of these parallel steps has to be removed from the tree, which means that its term argument has been reduced to a constructor term c (including *failure*), the parent of the step is reconsidered. Depending on c's value, either the parent term is reduced (to a *failure* if $c = \mathit{failure}$) and the other parallel steps are removed, or (if $c = \mathit{success}$) the computation of the other parallel steps continues normally.

3.12 Residuation

Residuation is a computational mechanism that delays the evaluation of a term containing an uninstantiated variable in a needed position [1]. Similar to narrowing, it supports the integration of functional programming with logic programming by allowing uninstantiated variables in functional expressions. However, in contrast to narrowing it is incomplete, i.e., unable to find all the solutions of some problems. Residuation is useful for dealing with built-in types such as numbers [9]. Residuation is meaningful only when a computation has several steps executing in parallel. If a computation has only one step executing, and this step residuates, the computation cannot be completed and it is said to *flounder*.

Operations that residuate are called *rigid*, whereas operations that narrow are called *flexible*. A formal model for the execution of programs defining both rigid and flexible operations is described in [15]. Our implementation already has the necessary infrastructure to accommodate this model. When a step s residuates on some variable V, we store (a reference to) s in V, mark s as *residuating* and continue the execution of the other steps. When V is bound, we remove the *residuating* mark from s so that s can be executed as any other step. If all the steps of a computation are *residuating*, the computation *flounders*.

4 The Compilation Process

The main motivation of this new implementation of narrowing is to provide a generic back end that can be used by functional logic languages based on a lazy evaluation strategy. Current work [6] shows that any narrowing computation in a left-linear constructor-based conditional rewrite system can be simulated, with little or no loss of efficiency, in an overlapping inductively sequential rewrite system, hence by our implementation. Therefore, our implementation can be used by languages such as Curry [20], Escher [23] and Toy [26].

To support this idea, our implementation works independently of any concrete source language. The source programs of our implementation are functional logic programs where all functions are defined at the top level (i.e., no local declarations) and the pattern-matching strategy is explicit. This language, called FlatCurry, has been developed as an intermediate language for the Curry2Prolog compiler [8] in the Curry development system PAKCS [16] and is used for various other purposes, e.g., meta-programming and partial evaluation [2]. Basically, a FlatCurry program is (apart from data type and operator declarations) a list of function declarations where each function f is defined by a single rule of the form $f(x_1, \ldots, x_n)$ = e, i.e., the left-hand side consists of pairwise different variable arguments and the right-hand side is an expression containing case expressions for pattern matching.

For instance, the function **leq** of Example 1 is represented in FlatCurry as follows (*fcase* denotes a case expression that is evaluated by narrowing):

$$
\begin{array}{l}
\texttt{leq(X,Y)} = \mathit{fcase}\ \texttt{X}\ \mathit{of}\ \{\ \texttt{zero} \to \texttt{true};\\
\qquad\qquad\qquad\qquad\quad \texttt{succ(M)} \to \mathit{fcase}\ \texttt{Y}\ \mathit{of}\ \{\texttt{zero} \to \texttt{false};\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{succ(N)} \to \texttt{leq(M,N)}\}\}
\end{array}
$$

A detailed description of FlatCurry including constructs for encoding features like non-deterministic choices (see Section 3.8), residuation (see Section 3.12), higher-order functions or conditional rules can be found on the Curry web page located at `http://www.informatik.uni-kiel.de/~curry/flat/`. Any inductively sequential program can be translated into FlatCurry rules whose right-hand side consists of only constructor/function applications and case expressions [17].

Although FlatCurry was originally designed as an intermediate language to compile and manipulate Curry programs, it should be clear that it can also be used for various other declarative languages (e.g., Haskell-like lazy languages with strict left-to-right pattern matching can be compiled by generating appropriate case expressions). To better accommodate a variety of source languages, our back end accepts a syntactic representation of FlatCurry programs in XML format so that other functional logic languages can be compiled into this implementation-independent format. Some examples together with the DTD for the XML FlatCurry representation are available at `http://www.informatik. uni-kiel.de/~curry/flat/`.

Our compiler, which is fully implemented in Curry, reads an XML representation and compiles it into a Java program following the ideas described in Section 3. Recall that every function is represented by a subclass of *operation*. For each function, we define a method *expand* which will expand a function call according to its rules and depending on its arguments (Sections 3.9, 3.10).

To show the simplicity of our compiled code, we provide an excerpt of the *expand* method for **`leq`** in Figure 3 which is generated from the case expression given above. According to Section 3.9, we must decide whether **`leq(`**t_1, t_2**`)`** is a redex. This expression is a redex if t_1 is a variable (we must narrow) or **`zero`** (we apply the first rule). If t_1 equals **`succ(..)`**, we must do the same check for the second argument. If t_1 fails, so does **`leq`**. If t_1 is a function call, we must evaluate it first. For the sake of simplicity, we show pseudo-code, which reflects the basic structure and is very similar to the real Java code.

To use our back end for a functional logic language, it is only necessary to compile programs from this language to a XML representation according to the FlatCurry DTD. For instance, our compiler can be used as a back end for Curry since Curry programs can be translated into this XML representation with PAKCS [16]. Again, it is worth emphasizing that FlatCurry can encode more than just Curry programs or needed narrowing, because the evaluation strategy is compiled into the case expressions. For instance, FlatCurry is a superset of TFL, which is used as an intermediate representation for a Toy-like language based on the CRWL paradigm (Constructor-based conditional ReWriting Logic) [21].

The computation engine is designed to work with the *read-eval-print* loop typical of many functional, logic and functional logic interpreters. In our Java implementation, the computation engine and the read-eval-print loop are threads that interact with each other in a producer/consumer pattern. When a computed expression (value plus answer) becomes available, the computation engine notifies the read-eval-print loop while preserving the state of the narrowing space. The read-eval-print loop presents the results to the user and waits. The user may request further results or terminate the computation. If the user requests a new result, the read-eval-print loop notifies the computation engine to further search the narrowing space. Otherwise, the narrowing space is discarded.

Currently we provide a naive trace facility that is useful to debug both user code and our own implementation. Since the computations originating from a goal

```
expand (Computation comp) {
    term = comp.getTerm();            //  get the term from top of the stack
    X = term.getArg(0);               // get first argument
    Y = term.getArg(1);               // get second argument
    switch on kind of X               // case X of ...
    case variable:                    // do narrowing: bind to patterns
        X.bindTo(zero);
        spawn new computation for leq(zero,Y);
        X.bindTo(succ(M));
        spawn new computation for leq(succ(M),Y);
        comp.setExhausted();              // this computation is exhausted
    case constructor:                     // argument is constructor-rooted,
        switch on kind of constructor     // thus do pattern matching
        case zero:                        // apply first rule:
            term.update(true);            // replace term with true
        case succ:                        // case X of succ(M) → case Y of...
           recursive case for switching on Y
    case failure:                         // the needed subterm has failed,
        term.update(failure)              // thus leq fails, too
    case operation:                       // X is a function call, thus
        comp.pushOnStack(X);              // evaluate this call first
}
```

Figure 3. Simplified pseudo-code for the `expand` method of **leq**

are truly concurrent, as is necessary to ensure operational completeness, and since some terms are shared between computations, the trace is not always easy to read. Computations are identified by a unique *id*. We envision a tool, conceptually and structurally well separated from the computation engine, that collects the interleaved traces of all computations, separates them, and presents each trace in a different window for each computation. This tool may have a graphical user interface to select which computations to see and/or interact with.

5 Related work

In this section we discuss and compare other approaches to functional logic language implementation (see [13] for a survey). Our approach provides an operationally complete and efficient architecture for implementing narrowing which can potentially accommodate sophisticated concepts, e.g., the combination of narrowing and residuation, encapsulated search or committed choice. As some recent narrowing-based implementations of functional logic languages show, most implementations that include these concepts lack completeness or are inefficient.

One common approach to implement functional logic languages is the transformation of source functional logic programs into Prolog programs. This approach is favored for its simplicity since Prolog has most of the features of functional logic languages: logical variables, unification, and non-determinism implemented by backtracking. However, the challenge in such an implementation is the implementation of a sophisticated evaluation strategy that exploits the presence of functions in the source programs. Different implementations of this kind are compared and evaluated in [14] where it is demonstrated that needed narrowing is efficiently implemented in a (strict) language such as Prolog and that this implementation is superior to other narrowing strategies. Therefore, most of the newer proposals to implement functional logic languages in Prolog are based on needed narrowing [4, 8, 14, 25]. In contrast to our implementation of narrowing, all of these efforts are operationally incomplete (i.e., existing solutions might not be found due to infinite derivation paths) since they are based on Prolog's depth-first search mechanism. The same drawback also occurs in implementations of functional logic languages based on abstract machines (e.g., [10, 21, 24, 27]) since these abstract machines use backtracking to implement non-determinism.

An exception is the Curry2Java compiler [18] which is based on an abstract machine implementation in Java but uses independent threads to implement non-deterministic choices. If these threads are fairly evaluated (which can be ensured by specific instructions), infinite derivations in one branch do not prevent finding solutions in other branches. Our approach is more flexible since it does not depend on threads, but it can control to any degree of granularity the scheduling of steps in distinct computations. This eases the implementation of problem-specific search strategies at the top level, whereas Curry2Java is restricted to encapsulated search [19].

Our implementation is the subject of active investigation in several directions. Thus, we are not specifically concerned with its efficiency at this time. Rather, we are studying architectures that easily integrate concepts and ideas that have been proposed for functional logic programming. Efficiency is an important issue, though, and we expect that it will be a strong point of our implementation due to the direct translation into an imperative language without the additional control layers of an abstract machine. While we have attempted to select an efficient architecture, we have not paid much attention to detailed optimization of our implementation, and we do not expect top speed as long as we compile to Java. We performed only a limited number of benchmarks to get a feel for where we stand.

For the functional evaluation, we evaluated the naive reverse of a list of 1200 elements (400 only for comparing Curry2Java). To benchmark non-determinism we evaluated **`add x y =:= peano300`**, where **`peano300`** denotes the term encoding 300 in unary notation and the infix operator **`=:=`** denotes the strict equality with unification. This goal is solved by creating 301 parallel computations by narrowing on the **`add`** operation.

The two fastest available implementations of needed narrowing, to the best of our knowledge, are the Curry2Prolog compiler of the PAKCS system and the *Münster Curry Compiler (MCC)* [27]. The Curry2Java back end (C2J), included in the PAKCS system, is not as fast, but is the fastest available correct and complete implementation of needed narrowing. We have also compared our approach to a Java-based implementation of Prolog: Jinni [30] is the fastest engine in the naive reverse benchmark among the Java-based Prolog implementations compared in [12]. Table 1 shows execution times, in seconds, for simple benchmarks on a PIII-900 MHz Linux machine. These results show that our engine is currently the fastest *complete* implementation of narrowing.

In all likelihood, its speed is partially due to the elimination of the overhead paid by Curry2Java for computing with an abstract machine. In comparison with Jinni, we perform better in the $\mathbf{rev}_{1200}$ benchmark, where the number of reduction steps is more or less the same for needed narrowing and SLD-resolution. For the **`add`** benchmark, we evaluate the goal **`add(X,Y,peano300)`** in Jinni. Due to the rules for strict equality with unification, even an optimized implementation of needed narrowing will

Table 1. Execution times for simple benchmarks on several FLP engines

	Ours	C2J	MCC	PAKCS	Jinni
$\texttt{rev}_{400}$	0.69	2.6			
$\texttt{rev}_{1200}$	5.5	N/A	0.69	0.68	45.9
$\texttt{add}_{300}$	2.1	16.2	0.12	0.09	2.5

perform at least twice as many reduction steps for **`add x y =:= peano300`** as a SLD-resolution of **`add(X,Y,peano300)`**. However, we are still faster than Jinni in this benchmark, too. Curry2Prolog and MCC are faster than our approach by a factor 8 for **`rev`** and by factor 20 for **`add`**. This is to be expected. Backtracking-based implementations are simpler and faster because they sacrifice completeness. Additionally, Curry2Prolog is executed by the highly optimized SICStus Prolog compiler, and the abstract machine of MCC is written in C, while our implementation is executed by the JVM. We expect that if our implementation were optimized and/or coded in C, it would offer performance competitive with these incomplete systems while retaining completeness.

A factor of 8-20 speedup over Java for a C implementation is reasonable and supported by the results of [18]. The authors have shown that a C++ implementation of the Curry2Java abstract machine was more than 50 times faster than the same implementation in Java. We do not expect a similar improvement because we have already eliminated the interpretation layer of the abstract machine, and because the results of [18] were obtained with JDK 1.1 while we use JDK 1.3. The latter is more efficient. However, we are confident that there are still considerable opportunities for improving the efficiency of our implementation. We plan to work on this aspect, but only after resolving the architectural issues related to the inclusion of encapsulated search, which is a very interesting feature for modern functional logic languages [19]. Its integration could cause some changes in our backend, e.g., to distinguish between local and global variables, which is one important issue of encapsulated search. However, most of the structures needed (nested computations, fair scheduling, explicit control of computations etc.) are already available in our backend. Thus, we expect the integration of encapsulated search to cause only minor changes or extensions.

6 Conclusion

We described the architecture of an engine for functional logic computations. Our engine implements an efficient, sound and complete narrowing strategy, *INS*, and integrates this strategy with other features, e.g., residuation and and-parallelism, desirable in functional logic programming. Our implementation is operationally complete, easy to extend (e.g., by external resources like constraint libraries) and general enough to be used as a back end for a variety of languages. Although our work is still evolving, simple benchmarks show that it is the fastest complete implementation of narrowing currently available: it has strong potential for further improvement in both performance and functionality.

Our implementation and supporting material is available under the GNU Public License at `http://nmind.cs.pdx.edu`.

References

1. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.
2. E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. In *Proc. 5th Intl. Symposium on Functional and Logic Programming (FLOPS '01)*, pages 326–342. Springer LNCS 2024, 2001.
3. S. Antoy. Definitional trees. In *Proc. 3rd Intl. Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
4. S. Antoy. Needed narrowing in Prolog. Technical report 96-2, Portland State University, 1996.
5. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. Intl. Conference on Algebraic and Logic Programming (ALP '97)*, pages 16–30. Springer LNCS 1298, 1997.
6. S. Antoy. Constructor-based conditional narrowing. In *Principles and Practice of Declarative Programming, (PPDP'01)*. ACM Press, Sept. 2001.
7. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal ACM*, 47(4):776–822, 2000. Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.
8. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. 3rd Intl. Workshop on Frontiers of Combining Systems (FroCoS '00)*, pages 171–185. Springer LNCS 1794, 2000.
9. S. Bonnier and J. Maluszynski. Towards a clean amalgamation of logic programs with external procedures. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 311–326. MIT Press, 1988.
10. M.M.T. Chakravarty and H.C.R. Lock. Towards the uniform implementation of declarative languages. *Computer Languages*, 23(2-4):121–160, 1997.
11. P.H. Cheong and L. Fribourg. Implementation of narrowing: The Prolog-based approach. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic programming languages: constraints, functions, and objects*, pages 1–20. MIT Press, 1993.
12. E. Denti, A. Omicini, and A. Ricci. `tu`Prolog: A light-weight Prolog for Internet applications and infrastructures. In *Practical Aspects of Declarative Languages (PADL)*, pages 184–198. Springer LNCS 1990, 2001.
13. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
14. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth Intl. Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.
15. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
16. M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.3: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000. Available at `http://www.informatik.uni-kiel.de/~pakcs`.
17. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
18. M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 1999(6), 1999.
19. M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint Intl. Symposium PLILP/ALP '98)*, pages 374–390. Springer LNCS 1490, 1998.
20. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.

21. T. Hortala-Gonzalez and E. Ullan. An abstract machine based system for a lazy narrowing calculus. In *Proc. 5th Intl. Symposium on Functional and Logic Programming (FLOPS '01)*, pages 216–232. Springer LNCS 2024, 2001.
22. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–443. MIT Press, 1991.
23. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3):1–49, 1999.
24. R. Loogen. Relating the implementation techniques of functional and functional logic languages. *New Generation Computing*, 11:179–215, 1993.
25. R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. 5th Intl. Symposium on Programming Language Implementation and Logic Programming*, pages 184–200. Springer LNCS 714, 1993.
26. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proceedings of RTA '99*, pages 244–247. Springer LNCS 1631, 1999.
27. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji Intl. Symposium on Functional and Logic Programming (FLOPS '99)*, pages 100–113. Springer LNCS 1722, 1999.
28. A. Middeldorp. Call by need computations to root-stable form. In *Proc. 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 94–105, 1997.
29. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.
30. P. Tarau. Jinni. Available at `http://www.binnetcorp.com/Jinni/`, 2001.

Deductive Verification for Multithreaded Java

Erika Ábrahám-Mumm[1], Frank S. de Boer[2], Willem-Paul de Roever[1], and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
[2] Utrecht University, The Netherlands

Abstract The semantical foundations of *Java* [9] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [2,6,19]). The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sublanguages (see e.g. [14, 18,21]). This paper presents a proof system for *multithreaded* *Java* programs. Concentrating on the issues of concurrency, we introduce an abstract programming language $Java_{MT}$, a subset of *Java* featuring object creation, method invocation, object references with aliasing, and specifically concurrency.

The assertional proof system for verifying safety properties of $Java_{MT}$ is formulated in terms of *proof outlines* [17], i.e., of annotated programs where Hoare-style assertions [8,12] are associated with every control point.

1 The programming language $Java_{MT}$

$Java_{MT}$ is a multithreaded well-typed sublanguage of *Java*. Programs, as in *Java*, are given by a collection of classes containing instance variable and method declarations. *Instances* of the classes, i.e., *objects,* are dynamically created, and communicate via *method invocation,* i.e., synchronous message passing. As we focus on a proof system for the concurrency aspects of *Java*, all classes in $Java_{MT}$ are thread classes in the sense of *Java*: Each class contains a start-method that can be invoked only once for each object, resulting in a new thread of execution. The new thread starts to execute the run-method of the given object while the initiating thread continues its own execution.

For variables, we notationally distinguish between *instance* and *temporary* variables, where instance variables are always private in $Java_{MT}$. Instance variables x hold the state of an object and exist throughout the object's lifetime. Temporary variables u play the role of formal parameters and local variables of method definitions and only exist during the execution of the method to which they belong. Therefore these temporary variables represent the local state of a thread of execution. Table 1 contains the abstract syntax of $Java_{MT}$.

For the semantics, we only highlight a few salient aspects. The formalization as structural operational semantics is given in [1].

The behaviour of a program results from the concurrent execution of threads, each described by the call-chain of its method invocations, given as a stack of local configurations. Threads can be created via new and started by (the first) invocation of the start-method. The invocation of a method extends the call chain by creating a new local configuration. It is removed from the stack when returning from the method. *Java* offers a synchronization mechanism for the mutually exclusive execution of methods: *Synchronized* methods of an object can be invoked only if no other threads are currently executing any synchronized methods of the same object.

exp	$::= x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp,\ldots,exp)$	$e \in Exp_c^t$	expressions
$sexp$	$::= \mathsf{new}^{c'} \mid exp.m(exp,\ldots,exp) \mid exp.\mathsf{start}()$	$sexp \in SExp_c^t$	side-effect exp.
stm	$::= sexp \mid x := exp \mid u := exp \mid u := sexp$		
	$\mid\ \epsilon \mid stm; stm \mid \mathsf{if}\ exp\ \mathsf{then}\ stm\ \mathsf{else}\ stm$		
	$\mid\ \mathsf{while}\ exp\ \mathsf{do}\ stm \ldots$	$stm \in Stm_c$	statements
$modif$	$::= \mathsf{nsync} \mid \mathsf{sync}$		modifiers
$rexp$	$::= \mathsf{return} \mid \mathsf{return}\ exp$		
$meth$	$::= modif\ m(u,\ldots,u)\{\ stm; rexp\}$	$meth \in Meth_c$	methods
$meth_{\mathsf{run}}$	$::= modif\ \mathsf{run}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{run}} \in Meth_c$	run-method
$meth_{\mathsf{main}}$	$::= \mathsf{nsync}\ \mathsf{main}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{main}} \in Meth_c$	main-method
$class$	$::= c\{meth\ \ldots\ meth\ meth_{\mathsf{run}}\}$	$class \in Class$	class defn's
$class_{\mathsf{main}}$	$::= c\{meth\ \ldots\ meth\ meth_{\mathsf{run}}\ meth_{\mathsf{main}}\}$	$class_{\mathsf{main}} \in Class$	main-class
$prog$	$::= \langle class\ \ldots\ class\ class_{\mathsf{main}}\rangle$		programs

Table 1. Java$_{MT}$ abstract syntax

2 The proof system

This section sketches the assertional proof system formulated in terms of *proof outlines* [7,17], i.e., where Hoare-style pre- and postconditions [8,12] are associated with each program statement. The proof system has to accommodate for shared-variable concurrency, aliasing, method invocation, and dynamic object creation.

2.1 The assertion language

The underlying assertion language consists of two different levels: The local assertion language specifies the behaviour on the level of method execution, and is used to annotate programs. The global behaviour, including the communication topology of the objects, is expressed in the global language used in the cooperation test.

In the language of assertions, we introduce as usual a countably infinite set of *logical variables* with typical element z disjoint from the instance and the local variables occurring in programs. Logical variables are used as bound variables in quantifications and, on the global level, to represent the values of local variables.

Table 2 defines the syntax of the assertion language. *Local expressions* are expressions of the programming language possibly containing logical variables. *Local assertions* are standard logical formulas over local expressions, where unrestricted quantification is allowed for integer and boolean domains only. Quantification over objects is only allowed in a restricted form asserting the existence of an element or a subsequence of a given sequence. Restricted quantification involving objects ensures that the evaluation of a local assertion indeed only depends on the values of the instance and temporary variables. In deference to the local assertion language, quantification on the global level is allowed for all types. Quantifications over objects range over the set of *existing* objects only.

2.2 Proof outlines

To be able to reason about the communication mechanism of method invocations, we split each method invocation statement into the sequential composition of an output

$exp_l ::= z \mid x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp_l, \ldots, exp_l)$		$e \in LExp_c^t$	local expressions
$ass_l ::= exp_l \mid \neg ass_l \mid ass_l \wedge ass_l$			
$\quad \mid\ \exists z(ass_l) \mid \exists z \in exp_l(ass_l) \mid \exists z \sqsubseteq exp_l(ass_l)$		$p \in LAss_c$	local assertions
$exp_g ::= z \mid \mathsf{nil} \mid \mathsf{f}(exp_g, \ldots, exp_g) \mid exp_g.x$		$E \in GExp^t$	global expressions
$ass_g ::= exp_g \mid \neg ass_g \mid ass_g \wedge ass_g \mid \exists z(ass_g)$		$P \in GAss$	global assertions

Table 2. Syntax of assertions

and an input statement representing the invocation of the method and the reception of the return value.

Next, we augment the program by fresh *auxiliary* variables. Assignments can be extended to multiple assignments, and additional multiple assignments to auxiliary variables can be inserted at any point. We introduce three specific auxiliary variables id, lock, and started to represent information about the global configuration at the proof-theoretical level. The temporary variable id of type Object $\times$ Int stores the identity of the object in which the corresponding thread has begun its execution, together with the current depth of its stack. The auxiliary instance variable lock of the same type is used to reason about thread synchronization: The value $\perp$ states that no threads are currently executing any synchronized methods of the given object; otherwise, the value (α, n) identifies the thread which acquired the lock, together with the stack depth n, at which it has gotten the lock. The boolean instance variable started states whether the object's start-method has already been invoked.

Finally, we extend programs by *critical sections,* a conceptual notion, which is introduced for the purpose of proof and, therefore, does not influence the control flow. Semantically, a critical section expresses that the statements inside are executed without interleaving with other threads.

To specify invariant properties of the system, the transformed programs are *annotated* by attaching pre- and postconditions, formulated in the local assertion language, to all occurrences of statements. Besides that, for each class c, the annotation defines a local assertion I_c called *class invariant,* which refers only to instance variables, and expresses invariant properties of the instances of the class. Finally, the *global invariant* $GI \in GAss$ specifies properties of communication between objects. We require that for all qualified references $E.x$ in GI, all assignments to x in class c are enclosed in critical sections.

2.3 Proof system

The global behaviour of a *Java* program results from the concurrent execution of method bodies, that can interact by

- shared-variable concurrency,
- synchronous message passing for method calls, and
- object creation.

Apart from the *initial correctness*, meaning that the annotation is correct with respect to the initial configuration, the proof system is split into three parts. The execution of a single method body in isolation is captured by *local correctness* conditions that show the inductiveness of the annotated method bodies and which are standard.

Interaction via synchronous message passing and via object creation cannot be established locally but only relative to assumptions about the communicated values. These assumptions are verified in the *cooperation test.* The communication can take place within a single object or between different objects. As these two cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules used in [5] and in [15] for CSP.

Finally, the effect of shared-variable concurrency is handled, as usual, by the *interference freedom test,* which is modeled after the corresponding tests in the proof systems for shared-variable concurrency in [17] and in [15]. In the case of *Java* it additionally has to accommodate for reentrant code and the specific synchronization mechanism.

Local correctness A proof outline is *locally correct*, if the usual verification conditions [4] for standard sequential constructs hold: The precondition of a multiple assignment to instance and local variables must imply the postcondition after execution of the assignment. As output and return statements do not affect the state of the executing thread, their preconditions must directly imply their postconditions. Finally, the pre- and postconditions of all statements of a class are required to imply the class invariant.

The interference freedom test The conditions of the interference freedom test ensure the invariance of local properties of a thread under the activities of other threads. Since we disallow public instance variables in Java_{MT}, we only have to deal with the invariance of properties under the execution of statements within the same object. Containing only temporary variables, communication and object creation statements do not change the state of the executing object. Thus we only have to take assignments $\boldsymbol{y} := \boldsymbol{e}$ into account.

Satisfaction of a local property of a thread may clearly be affected by the execution of assignments by a *different* thread in the same object. If, otherwise, the property describes the *same* thread that executes the assignment, the only control points endangered are those waiting for a return value earlier in the current execution stack, i.e., we have to show the invariance of preconditions of receive statements. Especially, the interference freedom test has to take care of *reentrant* method calls.

The cooperation test Whereas the verification conditions associated with local correctness and interference freedom cover the effects of assigning side-effect-free expressions to variables, the *cooperation test* deals with method invocation and object creation. Since different objects may be involved, it is formulated in the global assertion language. Besides defining verification conditions that ensure the invariance of the global invariant, it specifies conditions under which properties, whose evaluation depend on communicated values, are satisfied. Those properties are given by the preconditions of method bodies, and by the postconditions of receive and object creation statements.

3 Conclusion

In this extended abstract we sketched an assertional proof method for a multithreaded sublanguage of *Java*. The *soundness* of our method is shown by a standard albeit tedious induction on the length of the computation. Proving its *completeness* involves the introduction of appropriate assertions expressing reachability and auxiliary *history variables*. The details of the proofs can be found in [1].

Currently we are developing in the context of the European Fifth Framework RTD project Omega and the bilateral NWO/DFG project MobiJ a front-end tool for the computer-aided specification and verification of *Java* programs based on our proof method. Such a front-end tool consists of an editor and a parser for annotating *Java* programs, and of a compiler which translates these annotated *Java* programs into corresponding verification conditions. A theorem prover (HOL or PVS) is used for verifying the validity of these verifications conditions. Of particular interest in this context is an integration of our method with related approaches like the LOOP project [11, 16].

More in general, our future work focusses on the formalization of full-featured multithreading, inheritance, and polymorphic extensions involving behavioral subtyping [3].

Acknowledgements We thank Ulrich Hannemann for discussions and comments on an earlier version of the paper.

References

1. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept: Soundness and completeness. Technical Report TR-ST-01-2, Lehrstuhl für Software-Technologie, Christian-Albrechts-Universität Kiel, 2001.
2. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS State-of-the-Art-Survey. Springer, 1999.
3. P. America. A behavioural approach to subtyping in object-oriented programming languages. Technical report 443, Phillips Research Laboratories, 1989.
4. K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.
5. K. R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.
6. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [2].
7. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods.* Cambridge University Press, 2001.
8. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
9. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
10. C. Hankin, editor. *Programming Languages and Systems: Proceedings of the 7th European Symposium on Programming (ESOP '98), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), (Lisbon, Portugal, March/April 1998)*, LNCS 1381. Springer, 1998.

11. J. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Hankin [10].
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [13].
13. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.
14. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
15. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
16. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.
17. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
18. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In Swierstra [20], pages 162–176.
19. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer, 2001.
20. S. Swierstra, editor. *Proceedings of the 8th European Symposium on Programming (ESOP '99)*, LNCS 1576. Springer, 1999.
21. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. submitted for publication, 2002.

Mechanized Verification of Imperative Programs and Partial Functions

Jürgen Giesl

LuFG Informatik II, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany
giesl@informatik.rwth-aachen.de

Abstract Since *induction* is the essential proof method for program verification, several *induction theorem provers* [2, 3, 9, 12, 13] have been developed which support mechanized program verification. However, these systems are mainly designed for verifying *functional* programs. This represents a major problem for their application in practice where imperative programs are used almost exclusively.

Therefore, we suggest a new methodology in order to use the existing induction theorem provers for verification of imperative programs: First, imperative programs are transformed automatically into the functional input language of the induction prover [7]. Compared to standard translation techniques, the programs resulting from our transformation are significantly easier to verify. Then, existing induction provers should be applied in order to verify the resulting functions.

However, in general the resulting functions are *partial*, even if the imperative program terminates for every input. Unfortunately, most techniques for induction theorem proving are unsound when dealing with partial functions. However, we show that by slightly restricting the application of these techniques, they can still prove partial correctness of partial functions [8]. This means that the existing induction theorem provers can also be used for imperative programs without major changes.

1 Imperative Programs and Induction Provers

To handle imperative programs, they first have to be translated into the functional input language of existing induction provers. As an example consider the following imperative program which computes the division of x and y in its result variable z. (Before the statement "$z := \mathsf{0}$;" the program ensures that $y > \mathsf{0}$ and that y divides x.) It uses a data type nat whose objects are built with the *constructors* 0 and $\mathsf{s} : \mathsf{nat} \to \mathsf{nat}$ (for the successor function). Moreover, it calls a subtraction function "$-$" which is implemented by an auxiliary algorithm.

$$\begin{array}{l} \quad \ldots \\ z := \mathsf{0}; \\ \textbf{while } x \neq \mathsf{0} \textbf{ do } \; x := x - y; \; z := \mathsf{s}(z); \; \textbf{od}; \end{array}$$

To translate this program into a functional one, every **while**-loop is transformed into a separate function, cf. [14]. The function while corresponding to the loop above checks if the loop-condition is satisfied (i.e., if $x \neq \mathsf{0}$). In this case, while is called recursively with the new values of x, y, and z. Otherwise, while returns the result variable z. The program consisting of both statements above corresponds to the function div.

$$\begin{array}{ll}
\textbf{function}\ \mathsf{while} : \mathsf{nat} \times \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} & \\
\mathsf{while}(x, y, z) = \textbf{if}\ \ x \neq 0 & \\
\qquad\qquad\qquad\ \ \textbf{then}\ \mathsf{while}(x - y, y, \mathsf{s}(z)) & \\
\qquad\qquad\qquad\ \ \textbf{else}\ \ z &
\end{array}$$

$$\begin{array}{l}
\textbf{function}\ \mathsf{div} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\
\mathsf{div}(x, y) \ = \ \mathsf{while}(x, y, 0)
\end{array}$$

We use a functional language with eager (call-by-value) evaluation strategy. Then the semantics of the original imperative program is equivalent to the semantics of the translated functional one.

The functions resulting from this translation are always tail recursive. However, verifying tail recursive functions is difficult, because their accumulator parameter is usually initialized with a fixed value, but this value is changed in recursive calls. For example, while's accumulator z is initialized with 0 in the function div, but it changes during the execution of while. Hence, to verify div we would like to prove statements about $\mathsf{while}(x, y, 0)$, but in order to succeed with the proof, these statements have to be *generalized* to conjectures about $\mathsf{while}(x, y, z)$ [10,11]. To avoid the need for generalizations we developed a technique that *transforms* functions like div and while, which are difficult to verify, into algorithms which are much more suitable for automated induction proofs.

This is a novel application area for program transformations, because classical transformations [1,4,15] aim to increase efficiency. Such transformations are unsuitable for our purpose, since a more efficient algorithm is often harder to verify than a less efficient easier algorithm. As the goals of the existing transformations are opposite to ours, a promising starting approach was to use classical transformations *in the reverse direction*. Such an application of transformations for the purpose of verification has rarely been investigated before.

Starting from this idea, we extended and modified the transformations substantially in [7], which resulted in an automatic transformation procedure to increase verifiability. While of course our transformations are not always applicable, they proved successful on a representative collection of tail recursive functions. In this way, correctness of many imperative programs can be proved *automatically* without loop invariants or generalizations.

The basic idea of our transformations is to move away the *context* around recursive accumulator arguments such that the accumulator is no longer changed in recursive calls. For example, the result $\mathsf{while}(x - y, y, \mathsf{s}(z))$ can be replaced by $\mathsf{s}(\mathsf{while}(x - y, y, z))$ by moving the context $\mathsf{s}(\ldots)$ of the accumulator z outside of while's recursive call. Then while's accumulator z is no longer changed, and thus, it can be eliminated by replacing all its occurrences by 0. So finally, we obtain the following transformed algorithm where we renamed while to div and where we used a formulation with pattern matching instead of "**if**".

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{function}\ \mathsf{div} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}} \\
\mathsf{div}(0, y) & = & 0 \\
\mathsf{div}(\mathsf{s}(x), y) & = & \mathsf{s}(\mathsf{div}(\mathsf{s}(x) - y, y))
\end{array}$$

2 Induction Proving for Partial Functions

The translation of imperative into functional programs often generates partial functions, even if the imperative program is defined for all inputs. The reason is that termination of **while**-loops may depend on their contexts. In our example, the **while**-loop is only entered if $y > 0$ and if y divides x. However, this restriction on x and y is not present in the functions while and div. Therefore although the function corresponding to the *whole* imperative program will be total, its auxiliary function div is partial (e.g., $\mathsf{div}(\mathsf{s}(0), 0)$ is not terminating).

For partial functions we can at most verify their *partial correctness*. For instance, suppose that the specification for div is

$$\forall n, m : \mathsf{nat} \quad \mathsf{div}(n, m) * m = n.$$

Then div is in fact *partially correct*, i.e., for all n and m, if evaluation of $\mathsf{div}(n, m)$ is defined, then $\mathsf{div}(n, m) * m = n$. To express partial correctness, we use a definedness function def, where for any ground term t, $\mathsf{def}(t)$ is true iff evaluation of t is defined. Otherwise, $\mathsf{def}(t)$ is not defined either. So the partial correctness statement for div is

$$\mathsf{def}(\mathsf{div}(n, m)) \;\rightarrow\; \mathsf{div}(n, m) * m = n, \tag{1}$$

where all formulas are implicitly universally quantified. Our aim is to show that such statements are *inductively true*, i.e., that they hold for all data objects n, m, where the semantics of the functions is given by their algorithms.

In the following we present the three basic rules usually applied in induction theorem provers. We demonstrate that unfortunately, these rules are only sound if all occurring functions are total. However, we show that by slightly modifying their prerequisites, it is also possible to use them for partial functions.

2.1 Induction

When proving a conjecture about an algorithm f, one of the main ideas used in induction theorem proving is to perform an induction according to the recursion structure of this algorithm f [2, 16, 17]. Since (1) contains occurrences of $\mathsf{div}(n, m)$ this suggests an induction w.r.t. the recursions of the algorithm div where n and m are used as induction variables. For that purpose we perform a case analysis according to the defining equations of div (i.e., n and m are instantiated by $\mathsf{0}$ and y and by $\mathsf{s}(x)$ and y, respectively). Moreover, in the recursive case of div one may assume that (1) already holds for the arguments $\mathsf{s}(x) - y$ and y of div's recursive call. So instead of (1) it is sufficient to prove the following induction base formula (IB) and the induction step formula (IH) $\rightarrow$ (IC) which states that the induction hypothesis (IH) implies the induction conclusion (IC).

$$\mathsf{def}(\mathsf{div}(\mathsf{0}, y)) \;\rightarrow\; \mathsf{div}(\mathsf{0}, y) * y = \mathsf{0} \tag{IB}$$
$$\mathsf{def}(\mathsf{div}(\mathsf{s}(x), y)) \;\rightarrow\; \mathsf{div}(\mathsf{s}(x), y) * y = \mathsf{s}(x) \tag{IC}$$
$$\mathsf{def}(\mathsf{div}(\mathsf{s}(x) - y, y)) \;\rightarrow\; \mathsf{div}(\mathsf{s}(x) - y, y) * y = \mathsf{s}(x) - y \tag{IH}$$

Thus, in this proof one uses an induction relation $\succ_{\mathrm{ind}}$ where $(t_1, t_2) \succ_{\mathrm{ind}} (r_1, r_2)$ holds iff evaluation of $\mathsf{div}(t_1, t_2)$ leads to the recursive call $\mathsf{div}(r_1, r_2)$.

However, induction proofs are only sound if the induction relation used is *well founded* (i.e., if there exists no infinite descending chain $q_1^* \succ_{\mathrm{ind}} q_2^* \succ_{\mathrm{ind}} \ldots$ w.r.t. the induction relation $\succ_{\text{ind}}$, where q_i^* are tuples of terms). Here, the well-foundedness of the induction relation corresponds to the termination of the algorithm div. But as we already noticed, div is not always terminating!

Indeed, inductions w.r.t. non-terminating algorithms like div must not be used in an unrestricted way. For example, by induction w.r.t. the non-terminating algorithm f with the defining equation $\mathsf{f}(x) = \mathsf{f}(x)$ one could prove *any* formula, e.g., false conjectures like $\neg x = x$.

Hence, in the existing induction theorem provers, totality of all functions is required. However, it turns out that for (1) an induction w.r.t. the recursions of div is nevertheless possible. For that purpose we modify the former non-well-founded induction relation $\succ_{\text{ind}}$ by only defining $(t_1, t_2) \succ_{\text{ind}} (r_1, r_2)$ if $\mathsf{div}(t_1, t_2)$ leads to the recursive call $\mathsf{div}(r_1, r_2)$ *and evaluation of* $\mathsf{div}(t_1, t_2)$ *is defined.* This restricted relation is well founded although div is not always terminating.

With this modified induction relation we still get the formulas (IB) and (IH) $\rightarrow$ (IC), but in addition we also obtain a formula for the case where $\mathsf{div}(t_1, t_2)$ is not defined. (This is an additional base case of the induction.)

$$\neg\mathsf{def}(\mathsf{div}(n, m)) \;\rightarrow\; (1) \tag{PC}$$

In other words, an induction w.r.t. div only proves Conjecture (1) for those inputs where div is defined. Hence, we have to prove the additional *permissibility conjecture* (PC) to show that the conjecture also holds for inputs where div is not defined. If (PC) is true, then the induction proof w.r.t. the partial function div is permitted. In most examples, the permissibility conjecture is a tautology (e.g., $\neg\mathsf{def}(\mathsf{div}(n, m))$ contradicts the premise of (1)).

Hence, the successful rule for induction w.r.t. an algorithm f can now be extended to partial functions. The only modification needed is that for every application of the rule one has to prove an additional permissibility conjecture which checks whether the conjecture also holds if the algorithm f is not defined. Apart from inductions w.r.t. algorithms there is also a similar rule for *structural* inductions according to the definitions of data types.

2.2 Symbolic Evaluation

To continue our proof of (1), terms can be *symbolically evaluated* (i.e., defining equations of algorithms can be used as rewrite rules). For example, the first equation of div can be used to rewrite $\mathsf{div}(\mathsf{0}, y)$ to $\mathsf{0}$ in the induction base formula (IB), which yields $\mathsf{def}(\mathsf{0}) \rightarrow \mathsf{0} * y = \mathsf{0}$.

However, unrestricted symbolic evaluation would be unsound for partial functions. Due to the eager evaluation strategy of the functional language, a defining equation $f(t) = r$ can only be applied to evaluate the term $\sigma(f(t))$ for some substitution σ if the argument $\sigma(t)$ is defined, i.e., if $\mathsf{def}(\sigma(t))$ holds. For example, the term $\mathsf{div}(\mathsf{0}, \mathsf{div}(\mathsf{s}(\mathsf{0}), \mathsf{0}))$ has an undefined argument $\mathsf{div}(\mathsf{s}(\mathsf{0}), \mathsf{0})$ and thus, it may not be evaluated to $\mathsf{0}$. Hence, when rewriting the term $\sigma(f(t))$ in a formula φ, one also has to prove the permissibility conjecture $\neg\mathsf{def}(\sigma(t)) \rightarrow \varphi$. (For functions with several parameters, the definedness function def is extended to tuples of arguments where $\mathsf{def}(t_1, \ldots, t_n)$ stands for $\mathsf{def}(t_1) \wedge \ldots \wedge \mathsf{def}(t_n)$.) So in our example, in order to evaluate $\mathsf{div}(\mathsf{0}, y)$ in the induction base (IB) one also has to prove the permissibility conjecture $\neg\mathsf{def}(\mathsf{0}, y) \rightarrow$ (IB).

2.3 First-Order Consequence

The last rule performs inferences in standard first-order logic. It states that it is sufficient to prove lemmata $\psi_1, \ldots, \psi_n$ instead of the original conjecture φ, if $Ax \cup \{\psi_1, \ldots, \psi_n\} \vdash \varphi$ can be shown by a first-order calculus. Here, one may also use suitable axioms Ax about the data structures (which state that different constructors yield different data objects, etc.).

However, when regarding partial functions, it is recommendable to extend Ax by additional axioms which describe how the definedness function def operates on algorithms and constructors. Thus, one should add the axioms $\mathsf{def}(f(x_1, \ldots, x_n)) \to \mathsf{def}(x_1, \ldots, x_n)$ for all algorithms f and $\mathsf{def}(c(x_1, \ldots, x_n)) = \mathsf{def}(x_1, \ldots, x_n)$ for all constructors c.

In our example, one can now prove all proof obligations in the induction base case and in the induction step, the proof obligations can be reduced to

$$\mathsf{def}(u - v) \;\to\; (u - v) + v = u,$$

which can be proved analogously by induction w.r.t. the partial algorithm "$-$".

3 Conclusion

The rules in Sect. 2.1–2.3 constitute a calculus which is also sound for partial functions. The only difference between this calculus and the rules typically used for induction theorem proving (with total functions) is the function def and an additional permissibility conjecture which has to be proved whenever induction or symbolic evaluation are applied. Hence, the existing induction provers can easily be extended to this calculus and in this way, these systems can be directly used to reason about partial functions. So the restriction of induction provers to total functions is unnecessary, because in order to perform partial correctness proofs one does not need any information about the termination behavior. Further refinements, details, and extensions of this result as well as a thorough comparison with related work on partiality can be found in [8].

Our extension of induction proving to partial functions can be used for programs where the domain cannot be determined automatically as well as for programs with undecidable domains (e.g., interpreters or theorem provers), cf. [6]. Partial functions also occur frequently in program schemes and specifications (see [5] for an adaptation of our approach in order to reason about Z-specifications).

But in particular, the extension of induction theorem proving to partial functions is necessary in order to apply induction provers for the verification of *imperative programs*. Building on this extension, we developed a new methodology for mechanized verification of imperative programs: First, imperative programs are automatically transformed into the functional input language of the existing induction provers (which often yields partial functions). Then the existing induction provers are applied for their verification.

References

1. F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer, 1982.

2. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
3. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.
4. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.
5. D. Duffy and J. Giesl. Closure induction in a Z-like language. In *Proceedings of the International Conference of Z and B Users (ZB 2000)*, LNCS 1878, pages 471–490, 2000.
6. J. Giesl. The critical pair lemma: A case study for induction proofs with partial functions. Technical Report IBN 98/49, TU Darmstadt, 1998. http://www-i2.informatik.rwth-aachen.de/giesl.
7. J. Giesl. Context-moving transformations for function verification. In *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR '99)*, LNCS 1817, pages 293–312, 2000. A collection of examples can be found in the corresponding technical report IBN 99/51, TU Darmstadt. http://www-i2.informatik.rwth-aachen.de/giesl.
8. J. Giesl. Induction proofs with partial functions. *Journal of Automated Reasoning*, 26(1):1–49, 2001.
9. D. Hutter and C. Sengler. INKA: The next generation. In *Proceedings of the 13th International Conference on Automated Deduction (CADE-9)*, LNAI 1104, 1996.
10. A. Ireland and A. Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming*, 9:225–245, 1999.
11. A. Ireland and J. Stark. On the automatic discovery of loop invariants. In *4th NASA Langley Formal Methods Workshop*, NASA Conf. Publ. 3356, 1997.
12. D. Kapur and M. Subramaniam. New uses of linear arithmetic in automated theorem proving by induction. *Journal of Automated Reasoning*, 16:39–78, 1996.
13. M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Trans. Software Engineering*, 23(4):203–213, 1997.
14. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3, 1960.
15. A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28:360–414, 1996.
16. C. Walther. Mathematical induction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 2.* Oxford University Press, 1994.
17. H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In *Proceedings of the 9th International Conference on Automated Deduction (CADE-9)*, LNCS 310, 1988.

Model Checking Erlang Programs – LTL-Propositions and Abstract Interpretation

Frank Huch

Institute of Computer Science
Christian-Albrechts-University of Kiel
Kiel, Germany
fhu@informatik.uni-kiel.de

Abstract We present an approach for the formal verification of Erlang programs using abstract interpretation and model checking. In previous work we defined a framework for the verification of Erlang programs using abstract interpretation and LTL model checking. The application of LTL model checking yields some problems in the verification of state propositions, because propositions are abstracted too. In dependence of the number of negations in front of a propositions in a formula they must be satisfied or refuted. We show how this can automatically be decided by means of the abstract domain.

The approach is implemented as a prototype and we are able to prove properties like mutual exclusion or the absence of deadlocks and lifelocks for some Erlang programs.

1 Introduction

Growing requirements of industry and society impose greater complexity of software development. Consequently understandability, maintenance and reliability cannot be warranted. This gets even harder when we leave the sequential territory and develop distributed systems. Here many processes run concurrently and interact via communication. This can e.g. yield problems like deadlocks or lifelocks. To guarantee the correctness of software formal verifiaction is needed.

In industry the programming language Erlang [1] is used for the implementation of distributed systems. In [7] we have developed a framework for abstract interpretations [4,10,14] for a core fragment of Erlang. This framework gurantees that the transition system defined by the abstract operational semantics (AOS) includes all paths of the standard operational semantics (SOS). Because the AOS can sometimes have more paths than the SOS, it is only possible to prove properties that have to be satisfied on all paths, like in linear time logic (LTL). If the abstraction satisfies a property expressed in LTL, then also the program satisfies it, but not vice versa. If the AOS is a finite transition system, then model checking is decidable [12, 15]. For finite domain abstract interpretations and an additional flow-abstraction [9] this finite state property can be guaranteed for Erlang programs which do not create an unbound number of processes and use only mailboxes of restricted size.

However, the application of LTL model checking to the AOS is not that straight forward, as the following example shows: We assume the abstract domain $\widehat{A} = \{\mathbf{even}, \mathbf{odd}, \mathbf{num}, \mathbf{?}\}$, in which **even** and **odd** represent all even respectively odd numbers, **num** represents all numbers and **?** represents all values. LTL is usually defined over state propositions. For convenient specification of system properties, we allow arbitrary values of Erlang as possible propositions. Hence, in the abstraction possible state propositions are values of $\widehat{A}$.

Then we may ask in which states does the proposition **num** hold? As a matter of course in states containing the proposition **num**. The values **even** and **odd** are more precise than **num** and in states with these propositions **num** also holds. The abstract value **?** also represents other values (e.g. lists). For safeness of the abstraction, in a state of the AOS only containing the proposition **?** the proposition **num** must not hold. Although some concretizations of **?** are numbers.

LTL formulas can also contain negation, which makes the verification more complicated: In which states does the property $\neg$**num** hold? Although **num** does not hold in a state only containing the proposition **?** the property $\neg$**num** does not hold in this state either. **?** represents arbitrary values including numbers. For safeness of our approach non of the values may fulfill this property.

In Section 5 of this paper we discuss how it can be decided if an abstract proposition holds in a state. Furthermore, we formalize the semantics of abstract propositions in Section 6 and show how there verification can be integrated in standard model checkers in Section 7. Therefore, we first define the syntax and sketch the semantics of a core fragment of Erlang in Section 2. The framework for the abstract interpretation is shortly introduced in Section 3 and LTL is introduced inn Section 4. Finally, we present a concrete verification in Section 8 and conclude in Section 9.

2 Core Erlang

2.1 Syntax and Informal Semantics

Let Σ be a set of predefined function symbols with arity. For example $\texttt{+}/2 \in \Sigma$. Let $Var = \{\texttt{X},\texttt{Y},\texttt{Z},\ldots\}$ be a set of variables and $Atoms$ a set of atoms, e.g. `1`, `2`, `fail`, `succ`, ... Let $\mathcal{C}$ be the set of Erlang constructor functions with arity:

$$\mathcal{C} = \{\texttt{[.|.]}/2, \texttt{[]}/0\} \cup \{\texttt{\{...\}}/n \mid n \in \mathbb{N}\} \cup \{a/0 \mid a \in Atoms\}, \tag{1}$$

a constructor for building lists, a constructor for the empty list, constructors for building tuples of any arity and the atoms as constructors with arity 0.
The set of constructor terms is defined as the smallest set $T_{\mathcal{C}}(S)$ such that:

$$S \subseteq T_{\mathcal{C}}(S) \quad \text{and} \quad c/n \in \mathcal{C},\, t_1,\ldots,t_n \in T_{\mathcal{C}}(S) \Longrightarrow c(t_1,\ldots,t_n) \in T_{\mathcal{C}}(S)$$

The syntax of Core Erlang programs is defined as follows:

$$\begin{array}{lll}
p & ::= & f\texttt{(}X_1\texttt{,}\ldots\texttt{,}X_n\texttt{) -> }e\texttt{.} \mid p\ p \\
e & ::= & \phi\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mid X \mid pat \texttt{ = } e \mid \texttt{self} \mid e_1\texttt{,}e_2 \mid e_1\texttt{!}e_2 \mid \\
 & & \texttt{case } e \texttt{ of } m \texttt{ end} \mid \texttt{receive } m \texttt{ end} \mid \texttt{spawn(}f\texttt{,}e\texttt{)} \\
m & ::= & p_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}e_n \\
pat & ::= & c\texttt{(}p_1\texttt{,}\ldots\texttt{,}p_n\texttt{)} \mid X
\end{array}$$

All defined functions of a program, extended with their arity, built the set $FS(p)$. ϕ/n is an abbreviation for $f/n \in FS(p)$, $F/n \in \Sigma$ and $c/n \in \mathcal{C}$. In every Core Erlang program a main function is defined: $\texttt{main}/0 \in FS(p)$.

We call the set of Core Erlang terms e $ET(\emptyset)$. The set $ET(S)$ is defined by adding the grammar rule $e\ ::=\ v \in S$ for Core Erlang terms.

Erlang is a strict functional programming language. It is extended with processes, that are executed concurrently. With $\texttt{spawn(}f, \texttt{[}a_1,\ldots,a_n\texttt{])}$ a new process can be created anywhere in the program. The process starts with the evaluation of $f(a_1, \ldots, a_n)$. If the second argument of `spawn` is not ground, then it is evaluated before the new process is created. The functional result of `spawn` is the process identifier (*pid*) of the newly created process.

With $p!v$ arbitrary values (including pids) can be sent to other processes. The processes are addressed by their pids (p). A process can access its own pid with the Erlang function `self/0`. The messages sent to a process are stored in a mailbox and the process can access them conveniently with pattern matching in a `receive`-statement. Especially, it is possible to ignore some messages and fetch messages from further behind. For more details see [1].

Example 1. We consider the following Core Erlang program:

```
main() -> DB = spawn(dataBase,[[]]), spawn(client,[DB]),
          client(DB).

dataBase(L) -> prop(top)
   receive
      {allocate,Key,P} ->
          prop({allocate,P}),
          case lookup(Key,L) of
             fail -> P!free,
                     receive
                        {value,V,P} -> prop({value,P}),
                                       dataBase([{Key,V}|L])
                     end;
             {succ,V} -> P!prop(allocated), dataBase(L)
          end;
      {lookup,Key,P} -> prop(lookup),
                        P!lookup(Key,L), dataBase(L)
   end.
```

All applications of the function `prop` introduce state propositions, which will be used for the formal verification of the database. At this point, they can be ignored. There semantics will be discussed in Section 3.1.

The program creates a database process holding a state in which the key-value pairs are stored. This database is represented by a list of tuples, each consisting of a key and a corresponding value. The interface of the database is given by the messages `{allocate,Key,P}` and `{lookup,Key,P}`. Allocation is done in two steps. First the key is received and checked. If there is no conflict, then the corresponding value can be received and stored in the database. This exchange of messages in more than one step has to guarantee mutual exclusion on the database, because otherwise it could be possible that two client processes send keys and values to the database and they are stored in the wrong combination. A client can be defined accordingly [7]. In Section 8 we will prove that the database combined with two accessing clients satisfies this property.

2.2 Formal Semantics

In [7] we presented a formal semantics for Core Erlang. In the following we will refer to it as standard operational semantics (*SOS*). It is an interleaving semantics over a set of processes Π. Formally, a process consists of a pid ($\pi \in Pid := \{\texttt{@n} \mid n \in \mathbb{N}\}$), a Core Erlang evaluation term ($e \in ET(T_\mathcal{C}(Pid))$) and a word over constructor terms, representing the mailbox ($\mu \in T_\mathcal{C}(Pid)^*$). For the definition of the leftmost innermost evaluation strategy, we use the technique of evaluation contexts [5]:

$$\begin{array}{l} E ::= [\,] \mid \phi\texttt{(}v_1\texttt{,}\ \ldots\ \texttt{,}v_i\texttt{,}E\texttt{,}e_{i+2}\texttt{,}\ \ldots\ \texttt{,}e_n\texttt{)} \mid E\texttt{,}e \mid p \texttt{ = } E \\ \qquad\quad \texttt{spawn(}f\texttt{,}E\texttt{)} \mid E\texttt{!}e \mid v\texttt{!}E \mid \texttt{case } E \texttt{ of } m \texttt{ end} \end{array}$$

Here v denotes an evaluated expression, E the subterm the redex is in and e and m the parts which cannot be evaluated. $[\,]$ is called the hole and marks the point for the next evaluation. We shall then write $E[e]$ for the context E with the hole replaced by e. The next step of the evaluation takes place here. Analogously to the Core Erlang Terms $ET(S)$ over a set S, we name the Core Erlang contexts $EC(S)$. The set S defines, the set of values: $v \in T_\mathcal{C}(S)$. In the SOS this is $S = T_\mathcal{C}(Pid)$ and will be replaced by the abstract domain in the abstraction.

The semantics is a non-confluent transition system. The evaluations of the processes are interleaved. Only communication and process creation have side effects to other processes. For the modeling of these actions two processes are involved. To give an impression of the semantics, we present the rule for sending a value to another process:

$$\frac{v_1 = \pi' \in Pid}{\Pi, (\pi, E[v_1\,\texttt{!}\,v_2], \mu)(\pi', e, \mu') \stackrel{!v_2}{\Longrightarrow} \Pi, (\pi, E[v_2], \mu)(\pi', e, \mu' : v_2)}$$

The value is added to the mailbox of the process π' and the functional result of the send action is the sent value.

3 Abstract Interpretation of Core Erlang Programs

In [7] we developed a framework for abstract interpretations of Core Erlang programs. The abstract operational semantics (AOS) yields a transition system which includes all paths of the SOS. In an abstract interpretation $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ for Core Erlang programs $\widehat{A}$ is the abstract domain, which should be finite for our application in model checking. The abstract interpretation function $\widehat{\iota}$ defines the semantics of predefined function symbols and constructors. Its codomain is $\widehat{A}$. Therefore it is for example not possible to interpret constructors freely in a finite domain abstraction. $\widehat{\iota}$ also defines the abstract behaviour of pattern matching in equations, `case`, and `receive`. Here the abstraction can yield additional non-determinism, because branches can get undecidable in the abstraction. Hence, $\widehat{\iota}$ yields a set of results, which defines possible successors. Furthermore, an abstract interpretation contains a partial order $\sqsubseteq$, describing which elements of $\widehat{A}$ are more precise than other ones. We do not need a complete partial order or a lattice, because we do not compute any fixed point. We just evaluate the operational semantics with this abstract interpretation. An example for an abstraction of numbers with an ordering of the abstract representations is: $\mathbb{N} \sqsubseteq \{v \mid v \leq 10\} \sqsubseteq \{v \mid v \leq 5\}$. It is more precise to know, that a value is $\leq$5, than $\leq$10 than any number. The last component of $\widehat{\mathcal{A}}$ is the abstraction function. $\alpha : T_\mathcal{C}(Pid) \longrightarrow \widehat{A}$ maps every concrete value to its most precise abstract representation. Finally, the abstract interpretation has to fulfill five properties, which relate an abstract interpretation to the standard interpretation [7]. They also guarantee that all paths of the SOS are represented in the AOS, for example in branching.

An example for these properties is the following:

(P1) For all $\phi/n \in \Sigma \cup C, v_1, \ldots, v_n \in T_C(Pid)$ and $\widetilde{v}_i \sqsubseteq \alpha(v_i)$ it holds that $\phi_{\widehat{\mathcal{A}}}(\widetilde{v}_1, \ldots, \widetilde{v}_n) \sqsubseteq \alpha(\phi_{\mathcal{A}}(v_1, \ldots, v_n))$.

It postulates, that evaluating a predefined function or a constructor on abstract values, which are representations of some concrete values yields abstractions of the evaluation of the same function on the concrete values. The other properties postulate correlating properties for pattern matching in equations, `case`, and `receive`, and the pids represented by an abstract value. More details and some example abstractions can be found in [7,8].

3.1 Semantics of Propositions

We defined the semantics of Core Erlang as a labeled transition system. We want to prove properties of the system with model checking. It would be possible to specify properties using the labels. Nevertheless, it is more convenient to add propositions to the states of this transition system. With these state propositions properties can be expressed more easily. We use Core Erlang constructor terms as possible state propositions, which is very natural for Erlang programmers.

For the definition of propositions we assume a predefined Core Erlang function `prop/1`. The operational semantics of the function `prop` is the identity. Hence, adding applications of `prop` does not effect the SOS nor the AOS. Nevertheless, as a kind of side–effect the state in which `prop` is evaluated has the argument of `prop` as state proposition. We mark this with the label prop in the AOS:

$$\frac{-}{\Pi, (\pi, E[\mathtt{prop}(v)], \mu) \overset{\mathsf{prop}}{\longrightarrow}_{\widehat{\mathcal{A}}} \Pi, (\pi, E[v], \mu)}$$

The valid propositions of a process and a state can be evaluated with the function prop:

Definition 1. (Proposition of processes and states)
The proposition of a process is defined with the function $\mathsf{prop}_{\widehat{\mathcal{A}}} : \widehat{Proc}_{\widehat{\mathcal{A}}} \longrightarrow \mathcal{P}(\widehat{A})$:

$$\mathsf{prop}_{\widehat{\mathcal{A}}}((\pi, E[e], \mu)) := \begin{cases} \{\widehat{v}\} & \text{, if } e = \mathtt{prop}(\widehat{v}) \text{ and } \widehat{v} \in \widehat{A} \\ \emptyset & \text{, otherwise} \end{cases}$$

The propositions of a state $\mathsf{prop}_{\widehat{\mathcal{A}}} : \widehat{State}_{\widehat{\mathcal{A}}} \longrightarrow \mathcal{P}(\widehat{A})$ are defined as the union of all propositions of its processes:

$$\mathsf{prop}_{\widehat{\mathcal{A}}}(\Pi) = \bigcup_{\pi \in \Pi} \mathsf{prop}_{\widehat{\mathcal{A}}}(\pi)$$

For both functions we use the name $\mathsf{prop}_{\widehat{\mathcal{A}}}$. The concrete instance of this overloading will be clear from the application of $\mathsf{prop}_{\widehat{\mathcal{A}}}$. We will also omit the abstract interpretation in the index, if it is clear from the context.

In Example 1 we have added four propositions to the database, which have the following meanings:

`top`	marks the main state of the database process
`{allocate,P}`	marks that the process with the pid `P` tries to allocate a key
`{value,P}`	marks that the process with the pid `P` enters a value into the database
`lookup`	marks a reading access to the database

In most cases propositions will be added in a sequence, as for example the proposition `top`. Defining propositions with the function `prop` it is also possible to mark existing (sub–)expressions as propositions. As an example we use the atom `allocated`, which is sent to a requesting client, as a proposition.

4 Linear Time Logic

The abstract operational semantics defines a transition system. We want to prove properties of this transition system using model checking. The properties are described in a temporal logic. We use *linear time logic* (*LTL*) [6] in which properties have to be satisfied on every path of a given transition system.

Definition 2. (Syntax of Linear Time Logic (LTL)) Let *Props* be a set of state propositions. The set of *LTL–formulas* is defined as the smallest set with:

- $Props \subseteq \text{LTL}$ — state propositions
- $\varphi, \psi \in \text{LTL} \Longrightarrow$
  - $\neg\varphi \in \text{LTL}$ — negation
  - $\varphi \wedge \psi \in \text{LTL}$ — conjunction
  - $X\varphi \in \text{LTL}$ — in the next state φ holds
  - $\varphi\ U\ \psi \in \text{LTL}$ — φ holds until ψ holds

An LTL–formula is interpreted with respect to an infinite path. The propositional formulas are satisfied, if the first state of a path satisfies them. The next modality $X\varphi$ holds if φ holds in the continuation of the path. Finally, LTL contains a strong until: If φ holds until ψ holds and ψ finally holds, then $\varphi\ U\ \psi$ holds. Formally, the semantics is defined as:

Definition 3. (Path Semantics of LTL) An infinite word over sets of propositions $\pi = p_0p_1p_2\ldots \in \mathcal{P}(Props)^\omega$ is called a path. A path π satisfiess an LTL–formula φ ($\pi \models \varphi$) in the following cases:

$$\begin{array}{rcll}
p_0\pi & \models & P & \text{iff } P \in p_0 \\
\pi & \models & \neg\varphi & \text{iff } \pi \not\models \varphi \\
\pi & \models & \varphi \wedge \psi & \text{iff } \pi \models \varphi \text{ and } \pi \models \psi \\
p_0\pi & \models & X\varphi & \text{iff } \pi \models \varphi \\
p_0p_1\ldots & \models & \varphi\ U\ \psi & \text{iff } \exists i \in \mathbb{N} : p_ip_{i+1}\ldots \models \psi \text{ and } \forall j < i : p_jp_{j+1}\ldots \models \varphi
\end{array}$$

Formulas are not only interpreted with respect to a single path. Their semantics is extended to Kripke Structures:

Definition 4. (Kripke Structure) $\mathcal{K} = (S, Props, \longrightarrow, \tau, s_0)$ with S a set of states, *Props* a set of propositions, $\longrightarrow\subseteq S \times S$ the transition relation, $\tau : S \longrightarrow \mathcal{P}(Props)$ a labeling function for the states, and $s_0 \in S$ the initial state is called a *Kripke Structure*. Instead of $(s, s') \in\longrightarrow$ we usually write $s \longrightarrow s'$.

A *state path of* $\mathcal{K}$ is an infinite word $s_0s_1\ldots \in S^\omega$ with $s_i \longrightarrow s_{i+1}$ and s_0 the initial state of $\mathcal{K}$. If $s_0s_1\ldots$ is a state path of $\mathcal{K}$ and $p_i = \tau(s_i)$ for all $i \in \mathbb{N}$, then the infinite word $p_0p_1\ldots \in \mathcal{P}(Props)^\omega$ is a *path of* $\mathcal{K}$.

Definition 5. (Kripke–Structure–Semantics of LTL) Let $\mathcal{K} = (S, Props, \rightarrow, \tau, s_0)$ be a Kripke structure. It satisfies an LTL–formula φ ($\mathcal{K} \models \varphi$) iff for all paths π of $\mathcal{K}$: $\pi \models \varphi$.

The technique of *model checking* automatically decides, if a given Kripke structure satisfies a given formula. For finite Kripke structures and the logic LTL model checking is decidable [12].

For the convenient specification of properties in LTL we define some abbreviations:

Definition 6. (Abbreviations in LTL)

$$\begin{array}{rcll}
ff & := & \neg P \wedge P & \text{the boolean value true} \\
tt & := & \neg ff & \text{the boolean value false} \\
\varphi \vee \psi & := & \neg(\neg\varphi \wedge \neg\psi) & \text{disjunction} \\
\varphi \to \psi & := & \neg\varphi \vee \psi & \text{implication} \\
F\ \varphi & := & tt\ U\ \varphi & \text{finally } \varphi \text{ holds} \\
G\ \varphi & := & \neg F \neg\varphi & \text{globally } \varphi \text{ holds} \\
F^{\infty}\varphi & := & G\ F\ \varphi & \text{infinitely often } \varphi \text{ holds} \\
G^{\infty}\varphi & := & F\ G\ \varphi & \text{only finally often } \varphi \text{ does not hold}
\end{array}$$

The propositional abbreviations are standard. $F\varphi$ is satisfied if there exists a position in the path, where φ holds. If in every position of the path φ holds, then $G\varphi$ is satisfied. The formulas φ, which have to be satisfied in these positions of the path are not restricted to propositional formulas. They can express properties of the whole remaining path. This fact is used in the definition of $F^{\infty}\varphi$ and $G^{\infty}\varphi$. The weaker property $F^{\infty}\varphi$ postulates, that φ holds infinitely often on a path. Whereas $G^{\infty}\varphi$ is satisfied, if φ is satisfied with only finitely many exceptions. In other words there is a position, from where on φ always holds.

For the verification of Core Erlang programs we use the AOS respectively the SOS of a Core Erlang program as a Kripke structure. We use the transition system, which is spawned from the initial state (`@0`,`main()`,()). As labeling function for the states we use the function `prop` from the previous section.

5 Abstraction of Propositions

We want to verify Core Erlang programs with model checking. The framework for abstract interpretations of Core Erlang programs guarantees, that every path of the SOS is also represented in the AOS. If the resulting AOS is finite, then we can use simple model checking algorithms to check, if it satisfies a property φ expressed in LTL. If φ is satisfied in the AOS, then φ also holds in the SOS. In the other case model checking yields a counter example which is a path in the AOS on which φ is not satisfied. Due to the fact that the AOS contains more paths than the SOS, the counter example must not be a counter example for the SOS. The counter path can be a valid path in the abstraction but not in the SOS. Therefore, in this case it only yields a hint, that the chosen abstraction is too coarse and must be refined.

The application of model checking seems to be easy, but proving state propositions some problems appear, as the following example shows:

Example 2. `main() -> prop(42).`

A possible property of the program could be F **42** (finally 42). To prove this property we use the AOS with an abstract interpretation, for instance the even–odd interpretation, which only contains the values **even** (representing all even numbers), **odd** (representing all odd numbers), and **?** (representing all values). With this abstraction the AOS yields the following transition system:

$$(\texttt{@0}, \texttt{main()}, ()) \longrightarrow (\texttt{@0}, \texttt{prop}(42), ()) \longrightarrow (\texttt{@0}, \texttt{prop}(\mathbf{even}), ())$$
$$\xrightarrow{\mathsf{prop}} (\texttt{@0}, \mathbf{even}, ())$$

Only the state $(\texttt{@0}, \texttt{prop}(\mathbf{even}), ())$ has a property, namely **even**. **42** is an even number, but it is not the only even number. Therefore, this property cannot be proven, because of safeness. For example, we could otherwise also prove the property F **40**.

It is only possible to prove properties, for which the corresponding abstract value exclusively represents this value. But it does not make much sense, to abstract from special values and express properties for these values afterwards. Therefore, we only use propositions of the abstract domain, like

$$F\ \mathbf{even} \quad \text{(finally even)}$$

In the AOS the state (`@0`,`prop(`**even**`)`,()) has the property **even**. Therefore, the program satisfies this property. Now we consider a more complicated example:

Example 3. `main() -> prop(84 div 2).`

This system satisfies the property too, because $(84 \div 2) = 42$. But in the even–odd abstraction we only get:

$$\begin{array}{c} (\texttt{@0},\texttt{main()},()) \\ \downarrow \\ (\texttt{@0},\texttt{prop(84 div 2)},()) \\ \downarrow \\ (\texttt{@0},\texttt{prop(}\mathbf{even}\texttt{ div 2)},()) \\ \downarrow \\ (\texttt{@0},\texttt{prop(}\mathbf{even}\texttt{ div }\mathbf{even}\texttt{)},()) \\ \downarrow \\ (\texttt{@0},\texttt{prop(?)},()) \\ \downarrow \mathsf{prop} \\ (\texttt{@0},\texttt{?},()) \end{array}$$

with $\mathsf{prop}((\texttt{@0},\texttt{prop(?)},()))= \{?\}$ and $\emptyset$ as propositions of the other states. The result of the division of two even values must not be even. In a safe abstraction we cannot be sure, that the property F **even** is satisfied. Hence, model checking must yield, that it does not hold. For instance, for the program

```
main() -> prop(42 div 2).
```

the AOS is similar, but the property is not satisfied $(42 \div 2 = 21)$.

Therefore, a property is satisfied in a state, if the property of the state is at least as precise, as the expected property:

$$p_0 p_1 \ldots \models \widetilde{v} \text{ iff } \exists \widetilde{v}' \in p_0 \text{ with } \widetilde{v} \sqsubseteq \widetilde{v}'$$

But this is not correct in all cases, as the following example shows. We want to prove that the program satisfies the property

$$\psi = G\neg\mathbf{even} \quad \text{(always not even)}$$

Therefore, one point is to check that the state (`@0`,`prop(?)`,()) models $\neg\mathbf{even}$. With the definition from above we can conclude:

$$(\texttt{@0},\texttt{prop(?)},()) \not\models \mathbf{even} \qquad \text{and hence} \qquad (\texttt{@0},\texttt{prop(?)},()) \models \neg\mathbf{even}.$$

But that is wrong, because in Example 3 the property is not satisfied. The SOS has the property 42, which is an even value.

The problem is the non–monotonic operation $\neg$. Considering abstraction, the equivalence

$$\pi \models \neg\varphi \text{ iff } \pi \not\models \varphi$$

does not hold! $\pi \not\models \varphi$ only means that $\pi \models \varphi$ is not safe. In other words, there can be a concretization, which satisfies φ, but we cannot be sure that it holds for all concretizations. Therefore, negation has to be handled carefully.

Which value of our abstract domain would fulfill the negated proposition $\neg\mathbf{even}$? Only the proposition **odd** does. The values **even** and **odd** are incomparable and no value exists, which is more precise than these two abstract values. This connection can be generalized as follows:

$$p_0 p_1 \ldots \models \neg\widetilde{v} \text{ if } \forall \widetilde{v}' \in p_0 \text{ holds } \widetilde{v} \sqcup \widetilde{v}' \text{ does not exist}$$

Note, that this is no equivalence anymore. The non–existence of $\widetilde{v} \sqcup \widetilde{v}'$ does only imply that $p_0 p_1 \ldots \models \neg\widetilde{v}$. It does not give any information for the negation $p_0 p_1 \ldots \models \widetilde{v}$. This (double) negation holds, if $\exists \widetilde{v}' \in p_0$ with $\widetilde{v} \sqsubseteq \widetilde{v}'$.

On a first sight refuting a proposition seems not to be correct for arbitrary abstract interpretations. Consider the abstract domain

$$\begin{tikzcd} \leq\mathbf{0} \arrow[dr, dash] & & \geq\mathbf{0} \arrow[dl, dash] \\ & \mathbf{num} & \end{tikzcd}$$

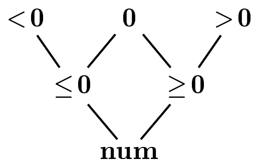
where the abstract values represent the following concrete values

abstract value	represented concrete values
$\leq\mathbf{0}$	$\{\mathbf{0}, \mathbf{-1}, \mathbf{-2}, \ldots\}$
$\geq\mathbf{0}$	$\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$
num	$\mathbb{Z}$

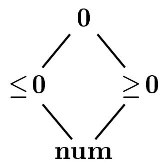
The represented concrete values of $\leq\mathbf{0}$ and $\geq\mathbf{0}$ overlap. Both represent the value $\mathbf{0}$. Therefore, it would be incorrect that a state with the proposition $\leq\mathbf{0}$ satisfies the formula $\neg\geq\mathbf{0}$.

But this abstract domain is not possible. The abstraction function $\alpha : A \longrightarrow \widehat{A}$ can only yield one abstract representation for a concrete value. Without loss of generality let $\alpha(\mathbf{0}) = \geq\mathbf{0}$. Abstract values, which represent the concrete value $\mathbf{0}$ can only be the result of the use of the abstract interpretation function $\widehat{\iota}$. But all these results $\widetilde{v}$ must be less precise: $\widetilde{v} \sqsubseteq \alpha(\mathbf{0}) = \geq\mathbf{0}$, because of the properties claimed by our framework. Hence, this abstract domain can be defined, but the value $\leq \mathbf{0}$ does only represent the values $\{-\mathbf{1}, -\mathbf{2}, \ldots\}$. The name of the abstract value is not relevant, but for understandability it should be renamed to $<\mathbf{0}$.

Alternatively, the abstract domain can be refined. The two overlapping abstract values can be distinguished by a more precise abstract value:



or



In both cases we must define $\alpha(\mathbf{0}) = \mathbf{0}$, because otherwise we have the same situation as before and the concrete value $\mathbf{0}$ is not represented by both abstract values $\leq \mathbf{0}$ and $\geq \mathbf{0}$.

6 Concretization of Propositions

With the advisement of the previous section we can now formalize, whether a proposition is satisfied respectively refuted. Similar results have been found in by Clark, Grumberg, and Long [3] and Knesten and Pnueli [11]. The result of Knesten and Pnueli introduces a solution to the problem informally, without any formalization. The paper of Clark et. al. formalizes a solution, but their framework differs from ours and the result cannot easily be transferd to our framework.

First we define the concretization of an abstract value. This is the set of all concrete values, which abstract to the value, or a more precise value.

Definition 7. (Concretization of Abstract Values)
Let $\widehat{\mathcal{A}} = (\widehat{A}, \iota, \sqsubseteq, \alpha)$ be an abstract interpretation. The concretization function $\gamma : \widehat{A} \longrightarrow \mathcal{P}(T_{\mathcal{C}}(Pid))$ is defined as

$$\gamma(\widetilde{v}) = \{v \mid \widetilde{v} \sqsubseteq \alpha(v)\}.$$

For the last example we get the following concretizations:

$$\begin{array}{lcl} \gamma(\mathbf{0}) & = & \{\mathbf{0}\} \\ \gamma(\leq\mathbf{0}) & = & \{\mathbf{0}, -\mathbf{1}, -\mathbf{2}, \ldots\} \\ \gamma(\geq\mathbf{0}) & = & \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\} \\ \gamma(\mathbf{num}) & = & \mathbb{Z} \end{array}$$

The following connections between the abstraction and the concretization function hold:

Lemma 1. *(Connections between γ and α)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \iota, \sqsubseteq, \alpha)$ be an abstract interpretation and γ the corresponding concretization function. Then the following properties hold:

1. $\forall v \in \gamma(\widetilde{v}) : \widetilde{v} \sqsubseteq \alpha(v)$
2. $\bigsqcap\{\alpha(v) \mid v \in \gamma(\widetilde{v})\} = \widetilde{v}$

Proof.

1. $v \in \gamma(\widetilde{v})$ iff $v \in \{v' \mid \widetilde{v} \sqsubseteq \alpha(v')\}$ iff $\widetilde{v} \sqsubseteq \alpha(v)$
2. $\bigsqcap\{\alpha(v) \mid v \in \gamma(\widetilde{v})\} = \bigsqcap\{\alpha(v) \mid v \in \{v' \mid \widetilde{v} \sqsubseteq \alpha(v')\}\} =$
   $\bigsqcap\{\alpha(v) \mid \widetilde{v} \sqsubseteq \alpha(v)\} = \widetilde{v}$

With the concretization function we can define, whether a proposition of a state satisfies a proposition in the formula or refutes it.

Definition 8. (Semantics of a Proposition)
Let $\widehat{\mathcal{A}} = (\widehat{A}, \iota, \sqsubseteq, \alpha)$ be an abstract interpretation. A set of abstract state propositions satisfies or refutes a proposition of a formula in the following cases:

$$\begin{array}{l} p \models \widetilde{v} \ \text{ if } \ \exists \widetilde{v}' \in p \text{ with } \gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v}) \\ p \not\models \widetilde{v} \ \text{ if } \ \forall \widetilde{v}' \in p \text{ holds } \gamma(\widetilde{v}) \cap \gamma(\widetilde{v}') = \emptyset \end{array}$$

With these definitions for the concretization we can define a corresponding definition for the abstract values. For finite domain abstractions they can be decided automatically.

Lemma 2. *(Deciding Propositions in the abstract domain)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \iota, \sqsubseteq, \alpha)$ be an abstract interpretation. A set of abstract state propositions satisfies or refutes a proposition of a formula in the following cases:

$$\begin{array}{l} p \models \widetilde{v} \ \textit{ if } \ \exists \widetilde{v}' \in p \textit{ with } \widetilde{v} \sqsubseteq \widetilde{v}' \\ p \not\models \widetilde{v} \ \textit{ if } \ \forall \widetilde{v}' \in p \textit{ holds } \widetilde{v} \sqcup \widetilde{v}' \textit{ does not exist} \end{array}$$

Proof. We show: $\widetilde{v} \sqsubseteq \widetilde{v}'$ implies $\gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v})$ and the non–existence of $\widetilde{v}' \sqcup \widetilde{v}'$ implies $\gamma(\widetilde{v}) \cap \gamma(\widetilde{v}') = \emptyset$:

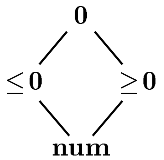
- $\widetilde{v} \sqsubseteq \widetilde{v}'$
  $\gamma(\widetilde{v}) = \{v \mid \widetilde{v} \sqsubseteq \alpha(v)\}$ and $\gamma(\widetilde{v}') = \{v \mid \widetilde{v}' \sqsubseteq \alpha(v)\}$.
  $(\widehat{A}, \sqsubseteq)$ is a partial order. Hence, it is transitive. This implies $\gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v})$
- $\widetilde{v} \sqcup \widetilde{v}'$ does not exist $\Longrightarrow \gamma(\widetilde{v} \sqcup \widetilde{v}') = \emptyset \Longrightarrow \{v \mid (\widetilde{v} \sqcup \widetilde{v}') \sqsubseteq \alpha(v)\} = \emptyset$
  $\Longrightarrow \{v \mid \widetilde{v}' \sqsubseteq \alpha(v) \text{ and } \widetilde{v} \sqsubseteq \alpha(v)\} = \emptyset \Longrightarrow \gamma(\widetilde{v}') \cap \gamma(\widetilde{v}) = \emptyset$

Note, that we only show an implication. We can define unnatural abstract domains, in which a property is satisfied or refuted with respect to Definition 8, but using only the abstract domain, we cannot show this. We consider the following abstract domain:

$$\begin{array}{c} \mathbf{0} \\ | \\ \mathbf{zero} \\ | \\ \mathbf{num} \end{array} \qquad \text{with } \alpha(v) = \begin{cases} \mathbf{0} & \text{, if } v = \mathbf{0} \\ \mathbf{num} & \text{otherwise} \end{cases}$$

The abstract value zero is superfluous, because it represents exactly the same values, as the abstract value $\mathbf{0}$. But this abstract domain is valid. Using the definition of the semantics of a proposition from Definition 8, we can show that $\{\mathbf{zero}\} \models \mathbf{0}$, because $\gamma(\mathbf{zero}) = \gamma(\mathbf{0}) = \{\mathbf{0}\}$. But $\mathbf{zero} \sqsubseteq \mathbf{0}$ and we cannot show that $\{\mathbf{zero}\} \models \mathbf{0}$ just using the abstract domain.

The same holds for refuting a proposition.



$$\text{with } \alpha(v) = \begin{cases} \geq\mathbf{0}\ , \text{ if } v \geq \mathbf{0} \\ \leq\mathbf{0} \text{ otherwise} \end{cases}$$

In this domain the abstract value $\mathbf{0}$ is superfluous. Its concretization is empty. Hence, $\gamma(\leq\mathbf{0}) = \{-\mathbf{1}, -\mathbf{2}, \ldots\}$ and $\gamma(\geq\mathbf{0}) = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$. $\gamma(\leq\mathbf{0}) \cap \gamma(\geq\mathbf{0}) = \emptyset$ and $\leq\mathbf{0} \models \neg\geq\mathbf{0}$. But this proposition cannot be refuted with this abstract domain, because $\leq\mathbf{0} \sqcup \geq\mathbf{0} = \mathbf{0}$ exists.

These examples are unnatural, because the domains contain superfluous abstract values. Nobody will define domains like these. Usually, the concretization of an abstract value is nonempty and differs from the concretizations of all other abstract values. In this case deciding propositions in the abstract domain is complete with respect to the semantics of propositions. Although it is not complete in general, it is safe. If we can prove a property with the abstract values, then it is also correct for its concretizations.

7 Proving LTL Formulas

So fare we have discussed, whether a proposition is satisfied or refuted. But in LTL negation is not only allowed in front of a proposition. Arbitrary sub–formulas can be negated. To solve this problem two different approaches are possible: In the first approach, all negations can be pushed inside the formula, until they only occur in front of the propositions. Therefore, we must extend LTL with the release modality $\varphi R \psi$, because there exists no equivalent representation of $\neg(\varphi U \psi)$, which uses negation only in front of φ and ψ. Release is the dual modality of until:

$$\neg(\varphi U \psi) \sim \neg\varphi R \neg\psi$$

Therefore, its semantics is defined as

$$p_0 p_1 \ldots \models \varphi R \psi \text{ iff } \forall i \in \mathbb{N} : p_i p_{i+1} \ldots \models \psi \text{ or } \exists j < i : p_j p_{j+1} \ldots \models \varphi$$

There is no intuitive semantics of release, except that it can be used for the negation of until. However, it can also be automatically verified in model checking.

Furthermore, we must add $\vee$ to LTL and use the following equivalences:

$$\neg\neg\varphi \sim \varphi\ , \quad \neg(\varphi \wedge \psi) \sim \neg\varphi \vee \neg\psi\ , \text{ and } \ \neg(X\varphi) \sim X\neg\varphi$$

With these equivalences we can push all negations into a formula and get an equivalent formula, in which negation only occurs directly in front of propositions. Then these formulas can be used for model checking. Positive and negative propositions can be checked with Lemma 2. Standard model checking algorithms work with a similar idea, but they do not need the release modality. For example, in [15] an alternating automaton is constructed, that represents the maximal model which satisfies the formula. The states correspond to the possible sub–formulas and their negations. For every negation in the formula the automaton switches to the corresponding state, which represents the positive respectively negative sub–formula. With this alternation the negations are pushed into the automaton representing the formula, like in the first approach. This leads to the second approach, in which a proposition has to be valued as a positive proposition, if it is used after an even number of negations. In the other case it is valued as a negative proposition. It has to be refuted.

We can use the same idea and distinguish two different kinds of propositions. The number of negations in front of a proposition are counted. In dependency of an even or an odd number of negations the propositions must be satisfied or refuted. It is possible, that the same property occurs more than ones in a formula. The different occurrences must be considered separately, because there can be different numbers of negations in front of them.

The advantage of this approach is, that we do not need the non–intuitive release modality. The formulas can be left as they are. The semantics of the propositions only depends on the number of negations in front.

The number of negations in front of a proposition can easily be computed with the following algorithm. We decent the formula inductively with a function mark. In a second argument mark accumulates if the number of negations in front of the actual sub–formula is even $(+)$ or odd $(-)$. If mark reaches a proposition, this proposition is annotated with the actual accumulated sign. If a negation occurs the algorithm flips $+$ and $-$.All other operators in the formula are just copied, without any modification. In the first call of mark no negations must be considered. Therefore, it is initially called with the sign $+$: For the two kinds of propositions we can now define

$$\begin{array}{l} p_0 p_1 \ldots \models^{+} \widetilde{v} \;\text{ if }\; \exists \widetilde{v}' \in p_0 \text{ with } \widetilde{v} \sqsubseteq \widetilde{v}' \\ p_0 p_1 \ldots \not\models^{-} \widetilde{v} \;\text{ if }\; \forall \widetilde{v}' \in p_0 \text{ holds } \widetilde{v} \sqcup \widetilde{v}' \text{ does not exist} \end{array}$$

8 Verification of the Database

Now we want to verify the system of Example 1. A database process and two clients are executed. We want to guarantee, that the process which allocates a key also sets the corresponding value:

> If a process π allocates a key, then no other process π' sets a value before π sets a value, or the key is already allocated.

This can for arbitrary processes be expressed in LTL as follows:

$$\bigwedge_{\substack{\pi \in Pid \\ \pi' \neq \pi}} G\ ({}^{-}\{\mathbf{allocate}, \pi\} \longrightarrow (\neg {}^{-}\{\mathbf{value}, \pi'\})\ U\ ({}^{+}\{\mathbf{value}, \pi\} \vee {}^{+}\mathbf{allocated}))$$

In our system only a finite number of pids occurs. Therefore, this formula can be translated into a pure LTL–formula as a conjunction of all possible permutations of possible pids, which satisfy the condition. This is

$$(\pi, \pi') \in \{(\texttt{@0}, \texttt{@1}), (\texttt{@0}, \texttt{@2}), (\texttt{@1}, \texttt{@2}), (\texttt{@1}, \texttt{@0}), (\texttt{@2}, \texttt{@0}), (\texttt{@2}, \texttt{@1})\}.$$

We have already applied the function mark to the formula. With respect to this marking the proposition ${}^{-}\{\mathbf{allocate}, \pi\}$ must be refuted. This is for example the case for the abstract values **top** and **{lookup,?}**. But **?** and $\{\mathbf{allocate}, p\}$ with p

the pid of the accessing client do not refute the proposition. The right side of the implication must be satisfied. Similar conditions must hold for the other propositions.

We can automatically verify this property using a finite domain abstraction, in which only the top–parts of depth 2 of the constructor terms are considered. The deeper parts of a constructor term are cut off and replaced by **?**. For more details see [8]. Our framework guarantees that the property also holds in the SOS and we have proven mutual exclusion for the database program.

9 Related Work and Conclusion

There exist two other approaches for the formal verification of Erlang: EVT [13] is a theorem prover especially tailored for Erlang. The main disadvantage of this approach is the complexity of proves. Especially, induction on infinite calculations is very complicated and automatization is not support yet. Therefore, a user must be an expert in theorem proving and EVT to prove system properties. We think for the practical verification of distributed systems push-button techniques are needed. Such a technique is model checking, which we use for the automatic verification of Erlang programs. This approach is also pursued by part of the EVT group in [2]. They verified a distributed resource locker written in Erlang with a standard model checker. The disadvantage of this approach is that they can only verify finite state systems. However, in practice many systems have an infinite (or for model checkers too large) state space. As a solution, we think abstraction is needed to verify larger distributed systems implemented in Erlang. Our approach for the formal verification of Erlang programs uses abstract interpretation and LTL model checking. The main idea is the construction of a finite abstract operational semantics with the use of a finite domain abstract interpretation. The abstraction is safe in the sense, that all path of the SOS are also represented in the AOS.

For convenient verification we have added propositions to the states of the AOS. Considering the abstract interpretation, problems in the semantics of these propositions arise. In this paper we solved these problems by distinguishing positive and negative propositions of the formula. For these we can decide, if a state satisfies or refutes it by means of the abstract domain. Finally, we used this technique in the formal verification of the database process: we proved mutual exclusion for two accessing clients.

References

1. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
2. Thomas Arts and Clara Benac Earle. Development of a verified Erlang program for resource locking. In *Formal Methods in Industrial Critical Systems*, Paris, France, July 2001.
3. Edmund Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
4. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

5. Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.
6. Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM SIGACT and SIGPLAN, ACM Press, 1980.
7. Frank Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).
8. Frank Huch. Verification of Erlang programs using abstract interpretation and model checking – extended version. Technical Report 99–02, RWTH Aachen, 1999.
9. Frank Huch. Model checking Erlang programs - abstracting the context-free structure. In *Proceedings of the Workshop on Software Model Checking*, volume 55–03 of *Electronic Notes in Theoretical Computer Science*, October 2001.
10. Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.
11. Yonit Kesten and Amir Pnueli. Modularization and abstraction: The keys to practical formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, *The 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS 1998)*, volume 1450 of *LNCS*, pages 54–71. Springer, 1998.
12. Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT-SIGPLAN, ACM Press.
13. Thomas Noll, Lars-åke Fredlund, and Dilian Gurov. The erlang verification tool. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 582–585. Springer, 2001.
14. David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. *LNCS*, 1503:351–380, 1998.
15. Moshe Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *LNCS*, pages 238–266. Springer, New York, NY, USA, 1996.

Automated Regression Testing of CTI-Systems

Andreas Hagerer[1], Tiziana Margaria[1], Oliver Niese[2], and Bernhard Steffen[2]

[1] METAFrame Technologies GmbH, Dortmund, Germany
{AHagerer,TMargaria}@METAFrame.de

[2] Chair of Programming Systems, University of Dortmund, Germany
{Oliver.Niese,Steffen}@cs.uni-dortmund.de

1 System-Level Testing of Complex Telephony Systems

The world of telecommunications has rapidly evolved during the last 15 years, modifying in this process its focus. In 1985 a telephone switch was 'only' used as a telephone switch. Additional components, either hardware or software, were gradually developed to bring additional functionality and flexibility to the traditional switch, e.g. in the initial days voice mail or billing systems. Today not only single functionalities are added at a quick pace, but the switch is mutating its role into the central element of complex heterogeneous and multivendor systems: it is nowadays integrated into whole business solutions, e.g. in the field of hotel solutions, call center and unified messaging applications. As a consequence, whereas in the earlier days the interaction between the switch and the applications was almost exclusively implemented via proprietary interfaces, the definition of open standards like e.g. *CSTA* or *TAPI* pushed the field towards the development of new, system level, applications. The left diagram in figure 1 documents the trend towards a growing product integration in terms of the increase in the number of value-added applications that work in average on or with a switch. As one can see, the integration factor is rapidly increasing since 1995, and the trend points in the direction of even larger value-added product ranges.

A parallel but concurring aspect is the increasing number of major switch releases per year (cf. figure 1 right). This trend is driven mainly by the convergence between the classical telecommunication technology and the modern IP technology, e.g. 'Voice-over-IP': modern switches are themselves complete complex systems, and they experience the accelerating evolution pace of hardware and software in combination!

A typical example of an integrated Computer-Telephony Integrated (briefly CTI) platfom is illustrated in figure 2, showing a midrange telephone switch and its environment. The switch is connected to the ISDN telephone network or, more generally, to the public switched telephone network (PSTN), and acts as a 'normal' telephone switch to the phones. Additionally, it communicates directly via a LAN or indirectly via an application server with CTI applications that are executed on PCs. Like the phones, CTI applications are active components: they may stimulate the switch (e.g. initiate calls), and they also react to stimuli sent by the switch (e.g. notify incoming calls). Therefore in a system-level test scenario it is necessary to investigate the interaction between such subsystems.

In the rapidly evolving scenario depicted above, the need for efficient automated regression testing is evident: whenever a release arises, either of the switch or of (a subset of) the application programs that cooperate with it,- singularly or, increasingly more often, in collaborative combinations - the correct functioning of the new configurations must be certified again.

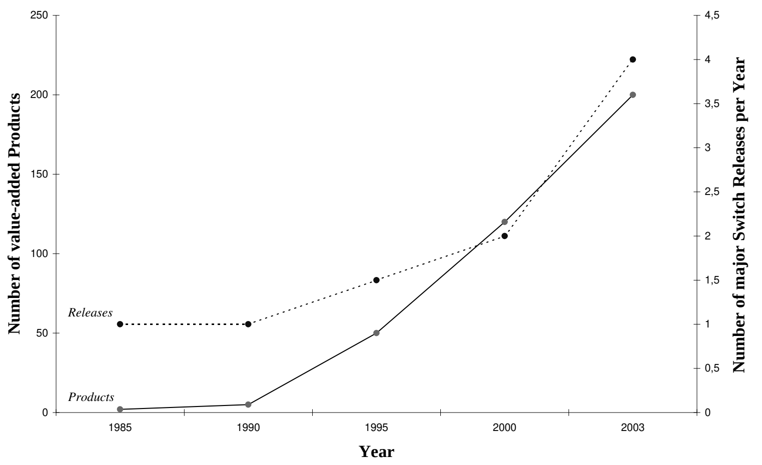


Figure 1. Trends in the development of CTI systems: (left) growing product integration and (right) faster paced switch releases

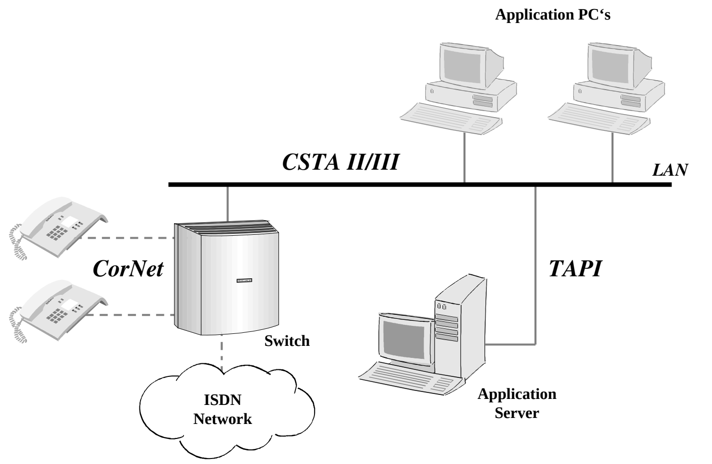


Figure 2. Example of an integrated CTI platform

Even the relatively simple scenario of figure 2 demonstrates the complexity of CTI platforms from the communication point of view, because there are several (internal) protocols involved. E.g. the telephones communicate via the *Corporate Network Protocol*[1] with the *Private Branch Exchange* (PBX), whereas the PBX communicates via *CSTA Phase II/III* protocol [1, 2] with the application server. On the application server, a *TAPI service provider* performs a mapping of the CSTA protocol to the TAPI protocol [8], which is the communication protocol between the application server and its clients.

This complex interplay between protocols must be considered when testing CTI systems, and it is clearly unfeasible to do this at the level of customary, fine grained protocol analysis.

[1] ECMA and CCITT Q.930/931 oriented D-channel layer 3 protocol for private IPABX

Additional scale complexity is introduced by the test tools themselves: for each significant interface (in the most convenient case covering a full participating subsystem, but more often for each device in a distributed setting) a specific, dedicated test tool participates in the regression test. Concretely, the simple scenario shown in figure 2 grows in the test laboratory to the dimensions shown in figure 4, whereby each of the devices and applications must be set up, steered, and reset during system level regression test.

Altogether, testing complex telephony solutions is a multidimensional task, which demands automation via adequate system level tool support. The complexity lies in the interaction between the components as well as in the short innovation cycles and the great number of possible combinations between the PBX and the value-added applications. Therefore an adequate environment must focus on structuring, efficiency and abstraction.

The paper is organized as follows: Section 2 describes our integrated test environment for system level regression testing of telephony systems. The successive sections describe the main features of our test environment: easy library-based design of test cases (section 3), reliable design of test cases (section 4), and the execution of test cases (section 5). Section 6 discusses the improvement of the usage of the integrated test environment while section 7 draws some conclusions.

2 The Integrated Test Environment

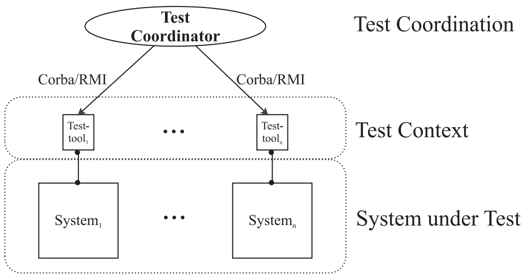


Figure 3. Architectural overview of the test environment

In summary systems-under-test have become composite (e.g. including *Computer Telephony Integrated* (CTI) platform aspects), embedded (due to hardware/software codesign practices), reactive, and run on distributed architectures (e.g. client/server architectures). Complex subsystems affect each other in a variety of complex ways, so mastering today's testing scenarios for telephony systems demands for an integrated, open and flexible approach to support the management of the overall test process, i.e. specification of tests, execution of tests and analysis of test results.

To handle the structural complexity, our approach offers a coarse grained testing environment, realized in terms of a component-based test design on top of a library of elementary but intuitively understandable test case fragments. The relations between the fragments are treated orthogonally, delivering a test design and execution environment enhanced by means of lightweight formal verification methods. This establishes a coarse-granular 'meta-level', on which

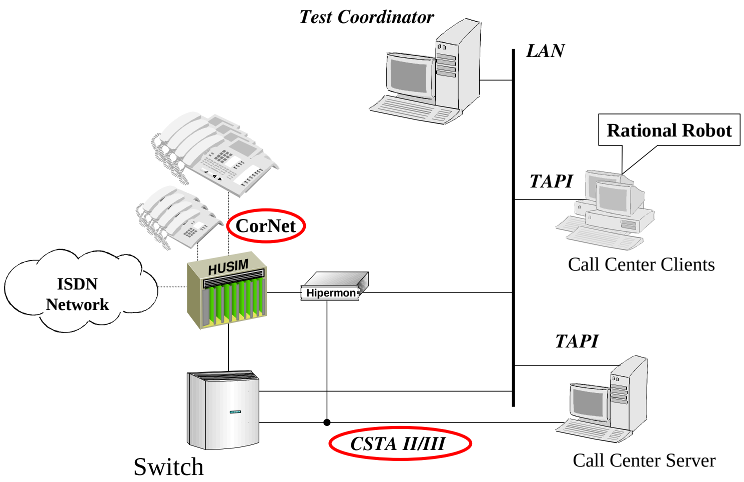


Figure 4. Concrete test setting

- test engineers are used to think,
- test cases can be easily composed,
- test suites can be configured and initialized,
- critical consistency requirements including version compatibility and frame conditions for executability are easily *formulated*, and
- consistency of test cases is fully automatically enforced via *model checking* and error diagnosis.

The *Integrated Test Environment* (*ITE*) is based on an existing general purpose environment for the management of complex workflows, METAFrame Technologies' *Agent Building Center* (ABC) [11], which contains built in features concerning test coordination and test organization. The currently available *Test Coordinator* (figure 3) constitutes the test management layer of our environment, and includes an application-specific specialization of the ABC for the domain of system level regression testing of telephony systems.

To communicate with different test tools, a flexible CORBA/RMI-based architecture has been designed for the *ITE*, cf. figure 3. The *Test Coordinator* executes integrated test cases by controlling several test tools, each managing its own subsystem. The extensibility of the environment by additional test tools is the key of the approach.

A concrete test scenario is shown in figure 4, which is used to test a complex call center solution. The call center consists of a switch with telephones of different kinds connected to a call center server, which controls several call center clients. In the considered scenario three different kind of test tools are supported by the test coordinator:

1. A proprietary protocol analyser (*Hipermon* [4]) which is connected to a telephone simulator (*Husim*) and to the connection between the switch and the application server.
2. A GUI test tool (*Rational Robot* [9]), which is used in several instances, i.e. for every considered call center client.

Additionally the test coordinator has access to the telephone switch itself, e.g. to perform an initialization at the beginning of a test case execution.

We now examine the main features of the environment, concerning easy, library-based design of test cases, the support for reliable design, and the support for test case execution in a heterogeneous distributed environment.

3 Design Support Features

System testing is characterized by focussing on inter-components cooperation. For the design of appropriate system-level test cases it is necessary to know what features the system provides, how to operate the system in order to stimulate a feature, and how to determine if features work. This information is gathered and after identification of the system's controllable and observable interfaces it is transformed into a set of stimuli (inputs) and verification actions (inspection of outputs, investigation of components' states). For each action a test block is prepared: a name and a class characterizing the block are specified and a set of formal parameters is defined to enable a more general usage of the block. In this way, for the CTI system to be tested a library of test blocks has been issued that includes test blocks representing and implementing, e.g.

Common actions Initialization of test tools, system components, test cases and general reporting functions,
Switch-specific actions Initialization of switches with different extensions,
Call-related actions Initiation and pick up of calls via a PBX-network or a local switch,
CTI application-related actions Miscellaneous actions to operate a CTI application via its graphical user interface, e.g., log-on/log-off of an agent, establish a conference party, initiate a call via a GUI, or check labels of GUI-elements.

The library of test blocks grows dynamically whenever new actions are made available.
The design of test cases consists in the behaviour-oriented combination of test blocks. This combination is done graphically, i.e., icons representing test blocks are graphically stuck together to yield test graph structures that embody the test behaviour in terms of control, see figure 5.

4 Verification Support Features

In our environment, the design of test cases is constantly accompanied by online verification of the global correctness and consistency of the test cases' control flow logic [6]. During the design phase, vital properties concerning the usage of parameters (local properties) and concerning the interplay between the stimuli and verification actions of a test case (global properties) can be verified. Design decisions that conflict with the constraints and consistency conditions of the intended system are thus immediately detected.

Local properties specified imperatively are used to check that the parameters are set correctly. Global properties concerning the interplay between arbitrarily distant test blocks of a test graph are expressed in a user-friendly specification language based

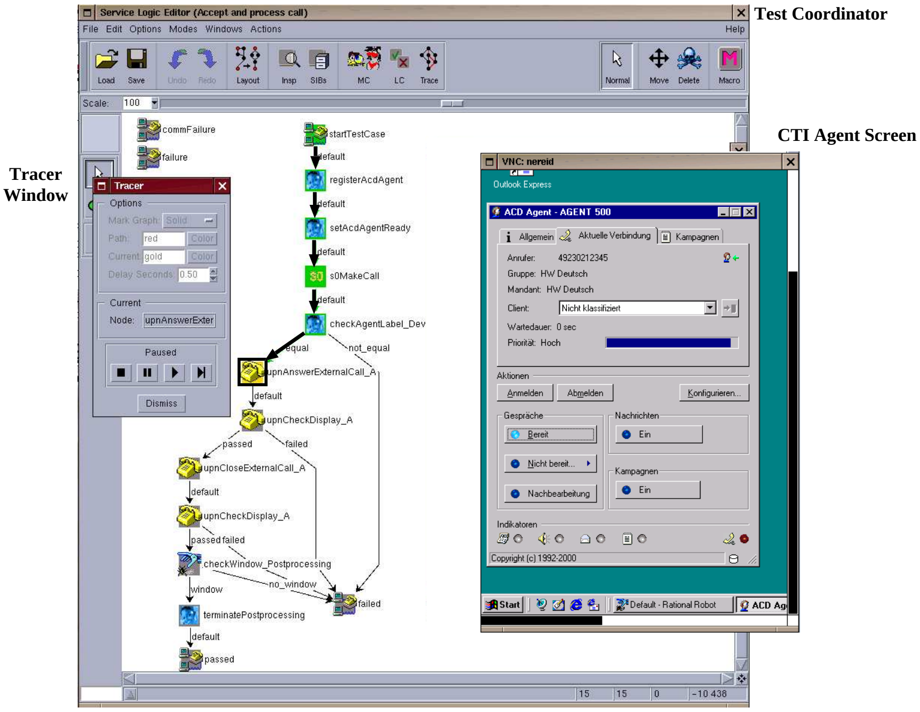


Figure 5. Test execution

on the Semantic Linear-time Temporal Logic [5], and are gathered in a constraint library accessed by the environment's model checker during verification. Typical constraints for the testing of CTI systems refer to the resource management, i.e. ensure that all used resources are freed after the test case execution[2].

If the model checker detects an inconsistency, a plain text explanation of the violated constraint appears. In addition, test blocks violating a local property as well as paths violating a global property are marked.

5 Execution Support Features

In the *Test Coordinator*, test cases can be executed immediately by means of ABC's tracer. Starting at a dedicated test block of a test graph the tracer proceeds from test block to test block. The actions represented by a test block are performed, i.e., stimuli and inspection requests are sent to the corresponding system's component, responses are received, evaluated, and the evaluation result is used to select one of the possibilities to pass control flow to a succeeding test block.

Figure 5 illustrates these features on a concrete test session snapshot. Here, the system-under-test is a call center application, in this case a client-server CTI application called "ACD Agent" which runs on different computers than the *Test Coordinator*. In this test we emulate a human call center agent with identifier *AGENT 500* and handle some actions via the agent's GUI of the PC application. The main window

[2] E.g. every hook-off for a device must be followed by a hook-on for this device.

of the *Test Coordinator* shows the actual executed test graph, where the execution path is highlighted. The execution can be controlled via the *Tracer Window*, e.g. a test graph can be executed either automatically or in a single step manner. The *CTI Agent Screen* shows the desktop of a call center agent, which is now controlled via the *Rational Robot*. The actions of the phones are controlled through the *Husim* and managed/observed through the *Hipermon*.

In general, the implementation of this execution scheme requires two activities during set-up of the *ITE*:

1. The actions referenced via test blocks have to be implemented by means of test tools. This task is performed by test engineers which are familiar with test tools, their handling and programming. For each action, the test engineers has to specify instructions to be executed by the test tool determined to support the specific action, e.g., via recording GUI-activities.
2. Specific tracer code has to be developed, that is assigned to the action's test block and that will be executed by the tracer. Experience with the CTI system shows that this code can be generated automatically for most actions. Manual development is necessary only if the test block shall initiate the execution of multiple actions in order to meet real-time requirements or if more complex evaluation of information about a component's state or reaction is required.

Finally, when executing a test graph, a detailed protocol is prepared. For each test block the tracer executes, all relevant data (its execution time, its name, the version of the files associated with the test block, the block's parameter values, and the processed data) are written to the protocol.

6 Evaluation

Table 1. Regression Test Cost factors

Task	manual	with ITE	frequency
Test planning	$\surd$	$\surd$	once
Test specification	$\surd$	$\surd$	once
Test scripts	($\surd$)	$\surd$	once
Test execution	$\surd$	-	recurrent
Test protocol	$\surd$	-	recurrent
Test analysis	$\surd$	($\surd$)	recurrent

To evaluate the economic impact of the *ITE* introduction, we must first identify the cost factors that pertain to testing CTI systems. Table 1 identifies the macroscopic cost factors that arise along the lifecycle. They are listed togehter with their frequency of occurrence and with a qualitative indication of their relevance in a manual testing and an automated testing scenario. Test planning and specification and the definition and setup of test scripts occur only initially, when an experimental scenario (i.e. the testing of a specific CTI system) is set up. The planning and specification phases are not affected by the *ITE*. The usual collection or programming of test scripts that in a manual setting directly constitute the elementary test blocks is in the *ITE* additionally supported by a largely automated generation of reusable test blocks that fit with the overall *ITE* architecture. The additional effort required by this wrapping is compensated by the increased reusal and ease of test design, but it requires in principle some additional effort.

The main focus of the *ITE* is however the reduction of costs for the repetitive, recurrent phases of CTI testing: primarily we address the test execution (cf. table 2), in combination with the automatic creation of test reports and (in near future) advanced support of the analysis of test results.

Table 2 documents the measured improvement of the test execution costs due to the introduction of *ITE*. The systems under tests considered in each row are composed by the PC client-server application listed in Col. 1 cooperating with the HICOM switch along the configuration pattern illustrated in figure 4. The second and third column report the measured effort (in man hours) of one regression cycle for the system under test when performed manually (Col. 2) or with the *ITE* (Col. 3). The improvement is dramatic: factors between 20 and 50 for each regression cycle execution. The full automation is for the moment not yet feasible since some manual steps like system setup and configuration (e.g. physical connection of the components, installation of the software on the machines) are still needed. These initial results are indeed representative for the average behaviour. Concerning the next applications that are joining the *ITE*, the conservative expectations shown in the last rows of table 2 indicate also a factor of about 40.

A global cost-benefit calculation shows that the additional investement for *ITE* is well able to pay off in a short period of time, if extensively adopted. *ITE* in fact dramatically reduces the recurring cost factors, without significatively increasing the remaining positions, that concern the basic effort that still has to be spent along the whole test lifecycle (test planning, manual configuration of the test settings, ...) and the necessary upfront investments (e.g. licence fees for test tools, hardware, ...).

Table 2. Test execution effort in hours per regression

System-under-test	manual	with ITE
Hotel Solutions	10,0	0,5
Call Center Solutions	43,0	1,0
Analog Voice Mail	23,0	0,5*
Digital Voice Mail	20,0	0,5*
Call Charge Computer	19,0	0,5*
Total	115,0	3,0

* Estimated values, since these systems are not yet fully integrated into the *ITE*.

7 Conclusion

We have implemented a formal methods-controlled, component-based test environment on top of a library of elementary but intuitively understandable test case fragments, in order to manage the increasing complexity of today's testing scenarios for telephony systems. The coarse-granular 'meta-level' established this way has proven to be adequate wrt. the way test engineers are used to think: Already after a few months of cooperation this coarse-granular test management support was successfully put into practice, drastically strengthening the pre-existing test environment.

We are not aware of any other test environment systematically addressing the needs of coordinating the highly heterogeneous test process, let alone on the basis of formal methods.

Being able to build on the *Agent Building Center*, which already contains features for the management of complex workflows, was a clear implementational advantage: we were able to demonstrate in a short time the practical satisfiability of the kernel requirements concerning test coordination and test organization. The *Test Coordinator*, which constitutes the test management layer of our environment, is already used in its current version in the test laboratories, and the test management has already proved to be capable of coordinating the different control and inspection activities of integrated system-level tests. Extensive use in the field has just begun and shows efficiency improvement of factors.

References

1. European Computer Manufactures Association (ECMA): *Services for Computer Supported Telecommunications Applications (CSTA) Phase II*, ECMA 217/218, 1994.
2. European Computer Manufactures Association (ECMA): *Services for Computer Supported Telecommunications Applications (CSTA) Phase III*, ECMA 269/285, 1998.
3. S. Gladstone: *Testing Computer Telephony Systems and Networks*, Telecom Books, 1996.
4. Herakom GmbH, Germany, `http://www.herakom.de`.
5. T. Margaria, B. Steffen: *Backtracking-free Design Planning by Automatic Synthesis in METAFrame*, Proc. FASE'98, Int. Conf. on Fundamental Aspects of Software Engineering, Lisbon, LNCS 1382, Springer Verlag, 1998, pp.188-204.
6. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, H.-D. Ide: *Library-based Design and Consistency Checking of System-level Industrial Test Cases*, Proc. FASE 2001, Int. Conf. on Fundamental Aspects of Software Engineering, LNCS 2029, Genova, Springer Verlag, 2001, pp.233-248.
7. O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, H.-D. Ide: *An Automated Testing Environment for CTI Systems Using Concepts for Specification and Verification of Workflows*, accepted for publication in Annual Review of Communic., Vol. 54, Int. Engineering Consortium, Chicago, 2000.
8. Microsoft Coorperation: *Using TAPI 2.0 and Windows to Create the Next Generation of Computer-Telephony Integration*, Whitepaper, `http://www.microsoft.com`.
9. Rational, Inc.: *The Rational Suite description*, `http://www.rational.com/products`.
10. B. Steffen, T. Margaria, V. Braun, N. Kalt: *Hierarchical Service Definition*, Annual Review of Communic., Vol. 51, Int. Engineering Consortium, Chicago, 1997, pp.847-856.
11. B. Steffen, T. Margaria: *METAFrame in Practice: Intelligent Network Service Design*, In *Correct System Design – Issues, Methods and Perspectives*, LNCS 1710, Springer Verlag, 1999, pp.390-415.

An Operational Procedure for the Model-Based Testing of CTI Systems

Andreas Hagerer[1], Hardi Hungar[1], Tiziana Margaria[1], Oliver Niese[2], and Bernhard Steffen[2]

[1] METAFrame Technologies GmbH, Dortmund, Germany
`{AHagerer,HHungar,TMargaria}@METAFrame.de`
[2] Chair of Programming Systems, University of Dortmund, Germany
`{Oliver.Niese,Steffen}@cs.uni-dortmund.de`

1 Moderated Regular Extrapolation

Moderated **regular extrapolation** aims at providing *a posteriori* descriptions of complex, typically evolving systems or system aspects in a largely automatic way. These descriptions come in the form of extended finite automata tailored for mechanically producing system tests, grading test suites and monitoring running systems. Regular extrapolation builds models from observations via techniques from machine learning and finite automata theory. These automatic steps are steered by application experts who observe the interaction between the model and the running system. This way, structural design decision are imposed on the model in response to the diagnostic information provided by the model generation tool in cases where the current version of the model and the system are in conflict.

Moderated regular extrapolation is particularly suited for *change management*, i.e. in cases where the considered system is steadily evolving, which requires continuous update of the system´s specification as well.

We will illustrate our method using a regression testing scenario for system level Computer Telephony Integration (*CTI*) [1]: Here, previous versions of the system serve as reference for the validation of future releases. A new release is required to support any unchanged feature and to enhance it with new or modified features. The iterative process of moderated regular extrapolation (Sec. 3) supports this system evolution, by incrementally building a model comprising the current spectrum of functionality on the basis of concise diagnostic feedback highlighting locations and sources of system/model mismatches.

2 The Computer/Telephony Scenario

Fig. 1 shows the considered scenario, a complex *Computer telephony integrated (CTI) system*, concretely a *Call center solution*. A midrange telephone switch is connected to the ISDN telephone network or, more generally, to the public switched telephone network (PSTN), and acts as a 'normal' telephone switch to the phones. Additionally, it communicates directly via a LAN or indirectly via an application server with CTI applications that are executed on PCs. Like the phones, CTI applications are active components: they may stimulate the switch (e.g. initiate calls), and they also react to stimuli sent by the switch (e.g. notify incoming calls).

The **model generator** is a novel part of the *Integrated Test Environment* (ITE for short) [2,3], in particular the *Test Coordinator*, an environment for the management

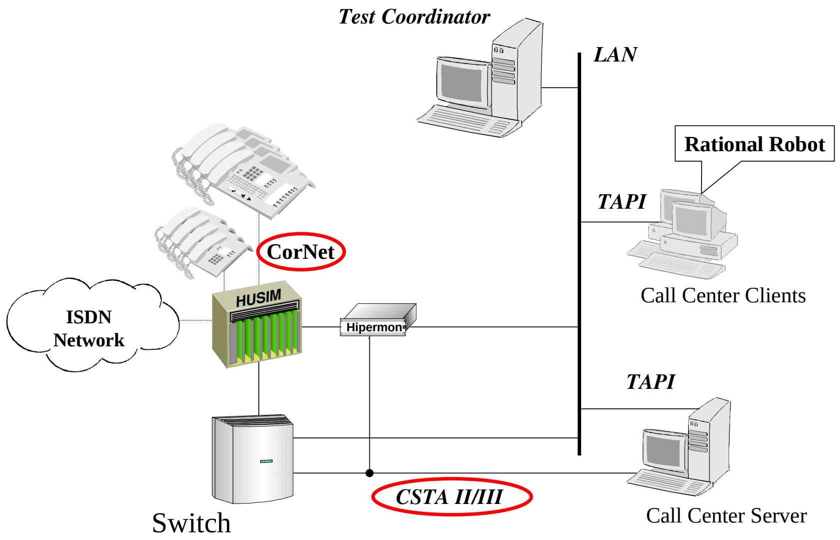


Figure 1. Overview of the Computer-Telephony Scenario

of the overall test process for complex systems, i.e. test specification, execution, and analysis of test runs.

3 Regular Extrapolation in Practice

In this section we sketch the approach, which will successively address the five steps of the model generation by regular extrapolation process by one simple example each.

3.1 Trace Collection

To build a model, the system is stimulated by means of test cases and the effects are traced and collected to form an initial model. Fig. 2 shows a simple test case as it is specified in the ITE by a test engineer. Here, three users pick up and hang up the handset of their telephones in arbitrary order. Test case executions are automatically protocolled in form of traces by the ITE's tracer (see e.g. Fig. 3). In a trace, both states and transitions are labeled with rich labels that describe portions of the system state and protocol messages respectively.

3.2 Abstraction

Here, we generalize observed traces to sequential behavioural patterns. The paper will illustrate the effect of abstracting from concrete components to *actors* playing specific roles. Fig. 3 shows an observed trace coming from the execution of the test case of Fig. 2 where this abstraction has taken place.

3.3 Folding

Folding a trace to a trace automaton allows a further powerful generalization of all possible interleaved combinations of actor-set traces. In the folding step, stable states

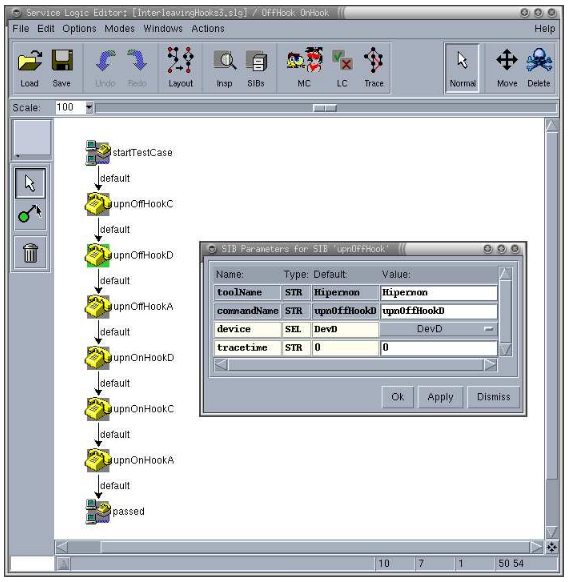


Figure 2. Example of a Test Graph

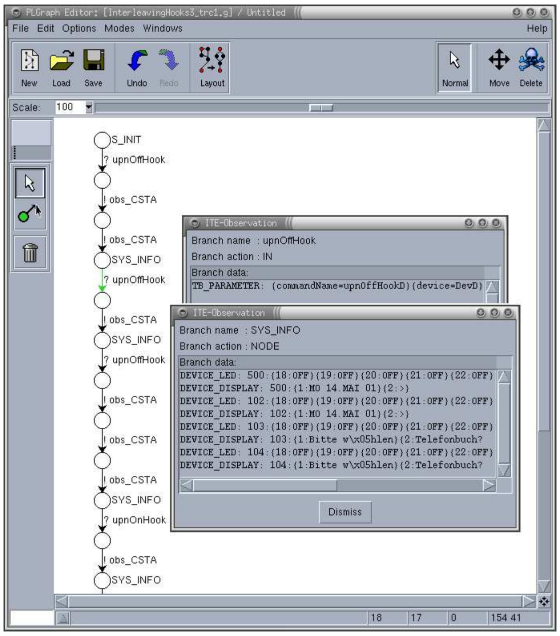


Figure 3. Example of an Observation Trace

that are considered equivalent are identified and can then be merged. For example, typically all observed devices are classified according to the status of display messages and LEDs. In this step *extrapolation* takes place: the behavior of the system observed so far is extrapolated to an automaton, which typically, due to cycle introduction, has infinite behavior.

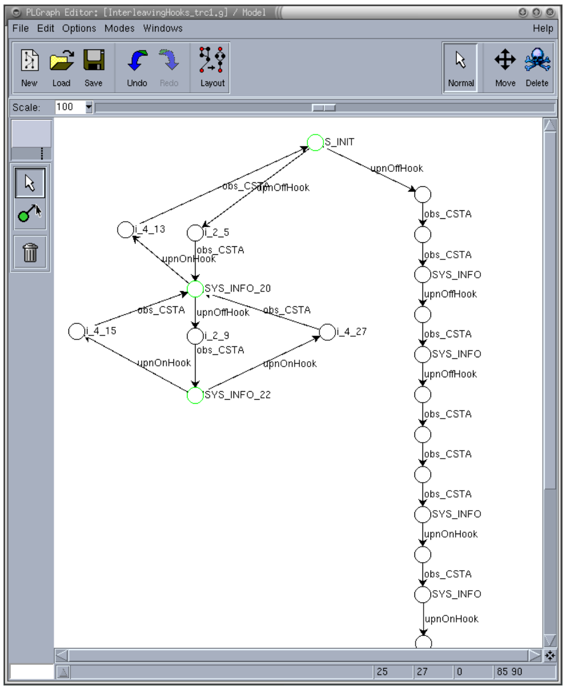


Figure 4. Adding a New Trace to the Model

The model shown in Fig. 4(left) has been generated via folding from a set of independent traces. It represents the behavior of two users picking-up and hanging-up handsets independently.

3.4 Refinement

With new observations, we can refine the model by adding further trace automata to a model. Again, each refinement step is based on the identification of behaviorally equivalent states. In Fig. 4 we show how the trace of Fig. 3, on the right, is added to the previous model on the left and leads to the model of Fig. 5 with four stable system states. Here, a system state is extremely abstract: it is characterized by the number of phones currently picked up. As a comparison, the observations on the original executable test cases were fully instantiated (e.g. they referred to single concrete device names).

3.5 Validation

Temporal properties of the models, reflecting expert knowledge, can be checked at any stage by means of standard model checking algorithms. This establishes an independent control instance: vital application-specific frame conditions, like safety criteria guaranteeing that nothing bad happens, or liveness properties guaranteeing a certain progress can automatically checked on the model. In case of failure, diagnostic information in terms of error traces reveals the source of trouble on the model level. The application then has to examine whether the revealed problem is just due

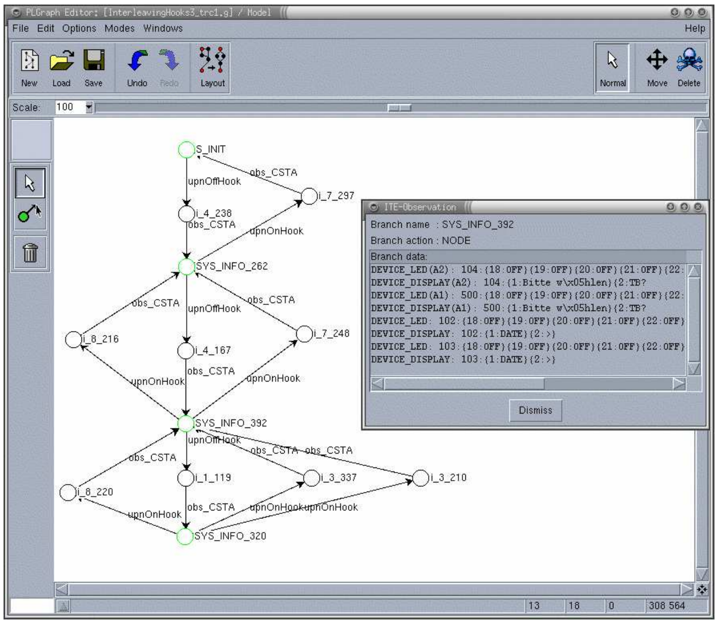


Figure 5. The Refined Optimized Model

to the inaccuracies of the model obtained so far, or whether there must be a problem in the underlying system as well.

Validation typically initiates the next iteration of the extrapolation process, which may now also involve technically more involved updating steps, like, e.g., model reduction, in cases where the model contained too many paths. Our system provides a number of automata theoretic operations and temporal synthesis procedures for the various updating steps. Moreover, it comprises algorithms for fighting the state explosion problem. This is very important, as already comparatively small sets of traces lead to quite big automata. E.g. Fig. 6 shows part of a model describing two very simple independent calls. For each call the model describes the correct interplay of the following actions: caller pick-ups handset, dials number, callee pick-ups handset, caller and callee hang-up their handsets. Already this simple scenario leads to a model with 369 states and 441 transitions.

References

1. A. Hagerer, T. Margaria, O. Niese, B. Steffen. Automated Regression Testing of CTI-Systems. In *Proc. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, K. Indermark and T. Noll (eds.), Aachener Informatik–Berichte N. 2001–11, Aachen University of Technology (RWTH), 2001, `ftp://ftp.informatik.rwth-aachen.de/pub/reports/2001/2001-11.ps.gz`,
2. O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, and H. Ide. An automated testing environment for CTI systems using concepts for specification and verification of workflows. *Annual Review of Communication*, Int. Engineering Consortium Chicago (USA), Vol. 54, pp. 927-936, IEC, 2001.

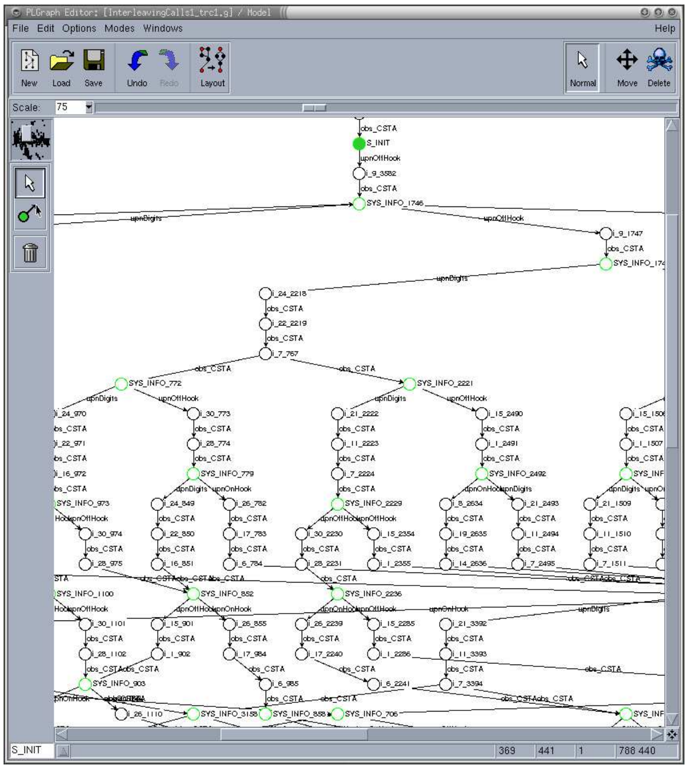


Figure 6. Example of a Larger POTS Model

3. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, and H. Ide. Library-based design and consistency checks of system-level industrial test cases. In H. Hußmann, editor, *Proc. FASE 2001*, LNCS 2029, pages 233–248. Springer Verlag, 2001.

Refining Stream Transformers to State-Based Components

Walter Dosch and Annette Stümpel

Institute for Software Technology and Programming Languages
Medical University of Lübeck
Ratzeburger Allee 160, D–23538 Lübeck, Germany
{dosch|stuempel}@isp.mu-luebeck.de

Abstract A black box specification of a deterministic software or hardware component refers to the function mapping input histories to output histories. An important refinement step amounts to designing a state-based component correctly implementing the specified behaviour. We present a formal method for refining stream transformers to state transition machines whose states arise from an abstraction of the input histories.

1 Introduction

Distributed systems are networks of components that communicate asynchronously via unidirectional channels. Streams model communication histories by recording the succession of messages on a channel. The input/output behaviour of a deterministic component is described by a stream processing function mapping input histories to output histories [5] . Stream processing functions, for a survey see [7], allow simple operators for serial composition, parallel composition, and feedback. The stream-based approach also supports modular refinement techniques, among others behavioural refinement, interface refinement, and communication refinement.

The top-down design starts with a black box specification which fixes the interface and the input/output behaviour without referring to the internal structure. The development process visualized in Fig. 1 leads to a network of elementary components suitable for a direct implementation.

An important design step consists in introducing a component's state space and in implementing the specified behaviour by a state transition system. The state-based description of a component prepares further refinement steps based on the structure and on the properties of the internal state.

In this paper, we present a formal method for transforming stream transformers into state-based components. In general, the output of the component will not only depend on the current input, but also on the previous input history. Therefore the state of the component must record the input history to the extent to which it influences the future output history. The state space results from an abstraction of the input histories; the functions operating on the state space can systematically be derived from the stream transformer. In summary, the state refinement of a stream transformer provides a correct implementation as a state-based component.

2 Streams and Stream Processing Functions

Streams model the temporal succession of messages on the channels between communicating components. The set of finite streams forms a *partial order* under the

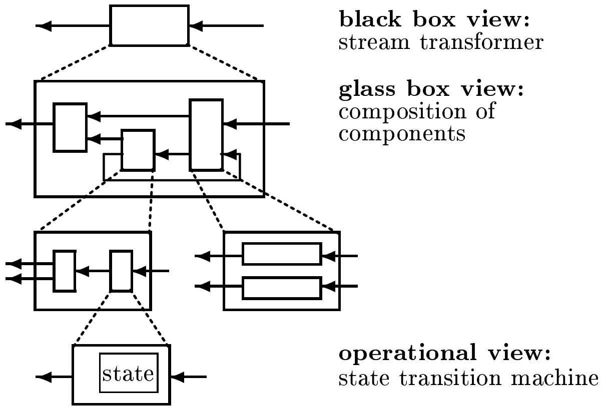


Figure 1. Refinement of distributed systems

prefix relation. It models operational progress in time: the shorter stream forms an initial part of the communication history.

A *stream processing function* maps an input stream to an output stream. It models a communicating component with one input and one output channel. The generalization to stream processing functions with several arguments and results is straightforward.

In the sequel, we concentrate on monotonic functions where further input leads to further output. A stream processing function is called *(prefix) monotonic*, for short a *stream transformer*, if any extension of the input history will result in an extension of the output history. Stream transformers incorporate a notion of causality since future input cannot cancel nor change previous output.

3 State Transition Machines with Input and Output

State transition machines with input and output are an abstract device modelling discrete state-based systems generating output driven by input.

A *state transition machine with input and output*, for short a *state transition machine*, consists of a nonempty set of *states*, an *input alphabet*, an *output alphabet*, a one-step *state transition function*, and a one-step *output function* mapping a state and an input to the successor state resp. to the finite output sequence. This type of state transition machine is useful for the high-level design of communicating components.

We extend the machine functions from a single input datum to a finite stream of input data. The *multi-step state transition function* yields the state transition effected by a finite input history. The *multi-step output function* yields the output history for a finite input history. For each state, the multi-step output function is prefix monotonic; so it describes a stream transformer. Two states of the machine are called *output equivalent* iff they generate the same stream transformer [4].

4 State Refinement

In this section, we present a formal method for refining stream transformers to state-based components. For simplicity, we confine ourselves to *strict* stream transformers generating no output for the empty input history. Given a suitable abstraction of the input history, we construct a state transition machine correctly implementing the stream transformer. Altogether, a *state refinement* associates with a stream transformer a state transition machine together with an initial state such that the original stream transformer agrees with the multi-step output function applied to the initial state.

4.1 Abstractions of the Input History

First we isolate the effect of the current input wrt. a previous input history. The *output extension* of a stream transformer denotes the prolongation of the output history effected by the current input after processing the input history. The output extension is well-defined, since the stream transformer is monotonic.

The state of a state transition machine collects relevant information from the input history that may influence the output on future input. An *abstraction function* associates with every input history a state representing the information needed. Two input histories may be identified under an abstraction function iff the stream transformer generates the same output for every future input stream no matter which of the two input histories has been processed before.

An abstraction function of input histories is required to be *transition closed*: if two input histories are identified, then their prolongations with the same input datum must be identified as well. Moreover, an abstraction function is called *output compatible* wrt. a stream transfomer, if it identifies at most input histories with the same output extension.

4.2 Constructing a State Transition Machine

Given a stream transformer and an output compatible abstraction function, we systematically construct a state transition machine correctly implementing the stream transformer, cf. Fig. 2.

stream transformer	$f : \mathcal{A}^\star \to \mathcal{B}^\star$
output extension	$\varepsilon : \mathcal{A}^\star \times \mathcal{A} \to \mathcal{B}^\star$
abstraction function	$\alpha : \mathcal{A}^\star \to Q$
state transition machine $M = (Q, \mathcal{A}, \mathcal{B}, \delta, \varphi)$	
initial state	$q_0 = \alpha(\langle\rangle)$
state transition function	$\delta(\alpha(X), x) = \alpha(X \,\&\, \langle x \rangle)$
output function	$\varphi(\alpha(X), x) = \varepsilon(X, x)$

Figure 2. State refinement of a stream transformer

The input and output alphabet $\mathcal{A}$ resp. $\mathcal{B}$ coincide with the types of the input and output streams of the stream transformer f. The set Q of states is the range of the abstraction function α. The initial state q_0 is the abstraction of the empty stream $\langle\rangle$. The state transition function δ associates with a state $\alpha(X)$ abstracting an input history X and a current input x the state respresenting the extended input history $X \,\&\, \langle x \rangle$. For a state $\alpha(X)$, the output function φ yields the output extension ε of the input history X and the current input x.

The state refinement is well defined, since the abstraction function is transition closed and assumed to be output compatible wrt. the stream transformer. The correctness of the state refinement is based on the following proposition: the multi-step output function of the constructed state transition machine applied to the initial state agrees with the original stream transformer.

For every stream transformer, the *canonical* state refinement uses the identity function as the abstraction of input histories. Then each state records the entire input history, and a state transition simply extends the input history. On the contrary, a *maximally reduced* state refinement identifies all output equivalent input histories.

5 Applications

We presented a formal method for systematically refining stream transformers to state-based components. The crucial design step consists in discovering useful abstractions of the input histories generating the state refinement.

As a first application, we consider an *iterator component* that repeatedly applies a basic function to all elements of the input stream. The iterator component processes the input datum by datum. The output only depends on the current input, but not on the previous input history. Therefore the iterator is a *history independent* component. The state refinement allows a maximally reduced state transition machine with a singleton state space.

As a second application, we consider a *scan component* having one input and one output port. The component produces the stream of proper prefixes of the input stream reduced under a binary operation. Two input histories are output equivalent iff their values reduced under the binary operation agree. An abstraction function leading to the maximally reduced state refinement reduces the input histories under the binary operation. The maximally reduced state transition machine of a *history sensitive* component manages with simple data values as states.

As a final application, we consider an input driven *shift register* which delays an input stream for a fixed number of steps. For sufficiently long input histories, the output history depends only on a fixed number of elements at the rear of the stream. Therefore the abstraction function that generates the maximally reduced state transition machine identifies all input histories which agree in the final segment of the specified length.

Further applications like memory components, control components or transmission components can be treated in quite a similar way.

6 Related Work

The approach is based on a general type of state transition machine with input and output. As related work, [1] investigate output equivalent states in Mealy machines which are specializations of generalized sequential machines. Lynch and Stark's port input/output automata [6] perform input, internal and output actions where in each state each input action is enabled. The ω-automata [8] are accepting devices for infinite streams having a finite number of states. Broy's group [2,3] uses state transition diagrams for verifying components of distributed systems. Here the transitions may depend on attributes of the previous and the successor state and also on the complete content of the input and output channels of the component before and after the transition.

References

1. I. Babcsányi and A. Nagy. Mealy-automata in which the output-equivalence is a congruence. *Acta Cybernetica*, 11(3):121–126, 1994.
2. Max Breitling and Jan Philipps. Step by step to histories. In T. Rus, editor, *Algebraic Methodology and Software Technology – AMAST'2000*, number 1816 in Lecture Notes in Computer Science, pages 11–25. Springer, 2000.
3. Manfred Broy. From states to histories. In Didier Bert, Christine Choppy, and Peter Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT'99*, number 1827 in Lecture Notes in Computer Science, pages 22–36. Springer, 2000.
4. Walter Dosch and Annette Stümpel. From stream transformers to state transition machines with input and output. In N. Ishii, T. Mizuno, and R. Lee, editors, *Proceedings of the 2nd International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'01)*, pages 231–238. International Association for Computer and Information Science (ACIS), 2001.
5. Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 471–475. North–Holland, 1974.
6. Nancy A. Lynch and Eugene W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82:81–92, 1989.
7. Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
8. Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3: Beyond Words, pages 389–455. Springer, 1997.

Some Applications of Goguen Categories in Computer Science

Michael Winter

Department of Computer Science
University of the Federal Armed Forces Munich
85577 Neubiberg, Germany
thrash@informatik.unibw-muenchen.de

Abstract Goguen categories were introduced as a suitable calculus for $\mathcal{L}$-fuzzy relations, i.e., for relations taking values from an arbitrary completely distributive lattice $\mathcal{L}$ instead of the unit interval $[0, 1]$ of the real numbers. In this paper we want to present three applications of such structures in computer science.

1 Introduction

One important application in computer science is the treatment of uncertain or incomplete information. To handle such kind of information, Zadeh [12] introduced the concept of fuzzy sets. Later on, Goguen [3] generalized this concept to $\mathcal{L}$-fuzzy sets and relations for an arbitrary completely distributive lattice $\mathcal{L}$ (or complete Brouwerian lattice) instead of the unit interval $[0, 1]$ of the real numbers.

Definition 1. Let $(\mathcal{L}, \sqcap, \sqcup, 0, 1)$ be a completely distributive lattice with meet $\sqcap$, union $\sqcup$, least element 0 and greatest element 1. Then the structure of $\mathcal{L}$-fuzzy relations is defined as follows:

1. A relation $Q : A \to B$ between sets A and B is function from $A \times B$ to $\mathcal{L}$.
2. For $Q : A \to B$ and $R : B \to C$ composition is defined by

$$(Q; R)(x, z) := \bigsqcup_{y \in B} (Q(x, y) \sqcap R(y, z)).$$

3. For $Q : A \to B$ conversion defined by $Q^{\smile}(x, y) := Q(y, x)$.
4. For $Q, S : A \to B$ join and meet are defined by

$$(Q \sqcup S)(x, y) := Q(x, y) \sqcup S(x, y), \qquad (Q \sqcap S)(x, y) := Q(x, y) \sqcap S(x, y).$$

5. The identity, the least and the greatest element are defined by

$$\mathbb{I}_A(x, y) := \begin{cases} 0 : x \neq y \\ 1 : x = y, \end{cases} \qquad \begin{array}{l} \perp\!\!\!\perp_{AB}(x, y) := 0, \\ \top\!\!\!\top_{AB}(x, y) := 1. \end{array}$$

An $\mathcal{L}$-fuzzy relation is called crisp iff $R(x, y) = 0$ or $R(x, y) = 1$ holds for all x and y. The structure defined above constitutes a Dedekind category introduced in [7].

Definition 2. A Dedekind category $\mathcal{R}$ is a category satisfying the following:

1. For all objects A and B the collection $\mathcal{R}[A,B]$ is a completely distributive lattice. Meet, join, the induced ordering, the least and the greatest element are denoted by $\sqcap, \sqcup, \sqsubseteq, \perp\!\!\!\perp_{AB}, \top\!\!\!\top_{AB}$, respectively.
2. There is a monotone operation $\smile$ (called conversion) such that for all relations $Q : A \to B$ and $R : B \to C$ the following holds: $(Q;R)^{\smile} = R^{\smile};Q^{\smile}$ and $(Q^{\smile})^{\smile} = Q$.
3. For all relations $Q : A \to B, R : B \to C$ and $S : A \to C$ the modular law $Q;R \sqcap S \sqsubseteq Q;(R \sqcap Q^{\smile};S)$ holds.
4. For all relations $R : B \to C$ and $S : A \to C$ there is a relation $S/R : A \to B$ (called the left residual of S and R) such that for all $X : A \to B$ the following holds: $X;R \sqsubseteq S \iff X \sqsubseteq S/R$.

Corresponding to the left residual, we define the right residual by

$$Q\backslash R := (R^{\smile}/Q^{\smile})^{\smile}.$$

This relation is characterized by $Q;Y \sqsubseteq R \iff Y \sqsubseteq Q\backslash R$.

We will use some basic properties of relations in a Dedekind category throughout the paper without mentioning. These properties and their proofs may be found in [1, 2, 8–10].

In some sense a relation of a Dedekind category may be seen as an $\mathcal{L}$-relation. The lattice $\mathcal{L}$ may equivalently be characterized by the ideal relations, i.e., a relation $J : A \to B$ satisfying $\top\!\!\!\top_{AA};J;\top\!\!\!\top_{BB} = J$, or by the scalar relations. A relation $\alpha : A \to A$ is called a scalar on A iff $\alpha \sqsubseteq \mathbb{I}_A$ and $\top\!\!\!\top_{AA};\alpha = \alpha;\top\!\!\!\top_{AA}$. The set of all scalars on A is denoted by $\mathrm{Sc}_{\mathcal{R}}(A)$. For $\mathcal{L}$-fuzzy relations the scalars are of the form $\begin{pmatrix} k & 0 & 0 \\ 0 & k & 0 \\ 0 & 0 & k \end{pmatrix}$ for an element $k \in \mathcal{L}$.

The notion of ideal elements was introduced by Jónsson and Tarski [4] and the notion of scalars by Furusawa and Kawahara [5].

In [11] it was shown that a suitable algebraic formalisation for arbitrary $\mathcal{L}$-fuzzy relations demands an extra operator. In particular, it was shown that there is no formula in the theory of Dedekind categories expressing the fact that a given $\mathcal{L}$-fuzzy relation is crisp. Therefore, the concept of Goguen categories was introduced. Our approach introduces two operations mapping every relation to the greatest crisp relation it contains resp. to the least crisp relation it is included in.

Definition 3. A Goguen category $\mathcal{G}$ is a Dedekind category together with two operations $^{\uparrow}$ and $^{\downarrow}$ satisfying the following:

1. $R^{\uparrow}, R^{\downarrow} : A \to B$ for all $R : A \to B$.
2. $(^{\uparrow}, ^{\downarrow})$ is a Galois correspondence, i.e., $S \sqsupseteq R^{\uparrow} \iff R \sqsubseteq S^{\downarrow}$ for all $R, S : A \to B$.
3. $(R^{\smile};S^{\downarrow})^{\uparrow} = R^{\uparrow\smile};S^{\downarrow}$ for all $R : B \to A$ and $S : B \to C$.
4. If $\alpha \neq \perp\!\!\!\perp_{AA}$ is a nonzero scalar then $\alpha^{\uparrow} = \mathbb{I}_A$.
5. For all antimorphisms[1] $f : \mathrm{Sc}_{\mathcal{G}}(A) \to \mathcal{G}[A,B]$ that $f(\alpha)^{\uparrow} = f(\alpha)$ for all $\alpha \in \mathrm{Sc}_{\mathcal{G}}(A)$ and all $R : A \to B$ the following equivalence holds

$$R \sqsubseteq \bigsqcup_{\substack{\alpha:A\to A \\ \alpha \text{ scalar}}} \alpha; f(\alpha) \iff (\alpha\backslash R)^{\downarrow} \sqsubseteq f(\alpha) \text{ for all } \alpha \in \mathrm{Sc}_{\mathcal{G}}(A).$$

[1] f is called an antimorphism iff $f(\bigsqcup M) = \bigsqcap_{\alpha \in M} f(\alpha)$ for all subsets M of $\mathrm{Sc}_{\mathcal{G}}(A)$

The obvious definition of $^\uparrow$ and $^\downarrow$ for $\mathcal{L}$-fuzzy relations

$$R^\uparrow(x,y) := \begin{cases} 1 \text{ iff } R(x,y) \neq 0 \\ 0 \text{ iff } R(x,y) = 0 \end{cases}, \qquad R^\downarrow(x,y) := \begin{cases} 1 \text{ iff } R(x,y) = 1 \\ 0 \text{ iff } R(x,y) \neq 1 \end{cases},$$

shows that this structure is a Goguen category.

According to the definitions above we call a relation R of an arbitrary Goguen category crisp iff $R^\uparrow = R$ (or equivalently $R^\downarrow = R$).

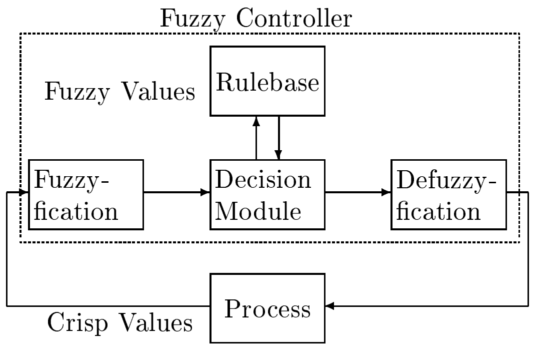
In [11] it was shown that the sets $\mathrm{Sc}_\mathcal{G}(A)$ of scalars on A are isomorphic via the mapping $\alpha \mapsto \mathbb{T}_{BA};\alpha;\mathbb{T}_{AB} \sqcap \mathbb{I}_B$.

The relation $(\alpha\backslash R)^\downarrow$ is called the α-cut of R. For $\mathcal{L}$-fuzzy relation it is characterized by $(\alpha\backslash R)^\downarrow(x,y)$ iff $R(x,y) \sqsupseteq \alpha$.

2 Fuzzy Controller

As a first application we want to show that fuzzy controller may be described by a simple term in the language of Goguen categories. This may be used to prove properties of the controller. A system like RelView developed at the Christian-Albrechts-University of Kiel or RATH developed at the Univesity of the Federal Armed Forces Munich is able to compute relational terms. Using such a system one easily gets a prototype of the controller.

We want to concentrate on the method of Mamdani (cf. [6]) for constructing a fuzzy controller. In this method a fuzzy controller consists of a rulebase, a decision module, a fuzzification and a defuzzification. This may be visualized by the following picture.

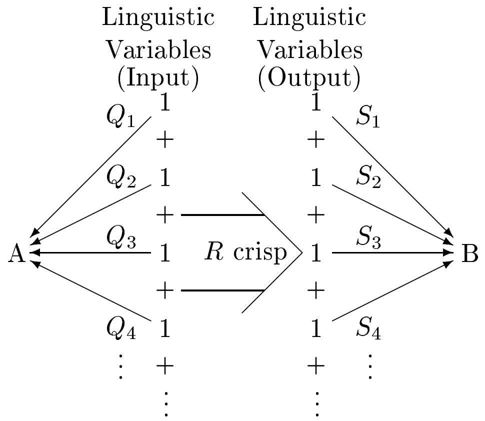


The rulebase is formulated using so-called linguistic variables, i.e.,abstract notions represented by common words from every day language (like: high speed, hot water, very heavy rain etc.). These linguistic variables are understood as meaning suitable fuzzy sets. Finally, a control rule is formulated as a conditional expression using the linguistic variables, e.g. there could be a rule like

$$\textbf{if } x \textbf{ is } \text{hot } \textbf{then } y = \text{negative small},$$

where x is a temperature and y is the input of a heating.

Such a fuzzy controller may be described within a Goguen category using the following picture.



The linguistic variables are modelled by a disjoint union of several copies of a one-element set. The relation R corresponds to the rulebase of the controller, and the relations Q_i and S_i are the fuzzy sets corresponding to every linguistic variable. The controller itself is described by the relational term

$$\mathcal{D}((\bigsqcup_{i\in I} Q_i^{\smile};\iota_i);R;(\bigsqcup_{j\in J}\iota_j^{\smile};S_j)),$$

where ι_i denotes the injection into the disjoined union, followed by a suitable defuzzification $\mathcal{D}$. The decision module is hidden in the composition operation ;. One may choose another composition operation induced by some t-norm like function on the underlying lattice.

As an example we want to construct a temperature controller. The input of the controller are temperatures taken from the intervall $[0,100]\subseteq\mathbb{Q}$ and the output is a value from $[-20,20]\subseteq\mathbb{Z}$, the adjusting values of the heating. The linguistic variables are given by

$$\begin{aligned}\mathrm{LV}_{\mathrm{in}} = \{&\mathrm{EC} = \text{extrem cold}, \\ &\mathrm{VC} = \text{very cold}, \\ &\mathrm{C} = \text{cold}, \\ &\mathrm{M} = \text{medium}, \\ &\mathrm{W} = \text{warm}, \\ &\mathrm{H} = \text{hot}, \\ &\mathrm{VH} = \text{very hot}\}.\end{aligned} \qquad \begin{aligned}\mathrm{LV}_{\mathrm{out}} = \{&\mathrm{NB} = \text{negative big}, \\ &\mathrm{NS} = \text{negative small}, \\ &\mathrm{ZO} = \text{zero}, \\ &\mathrm{PS} = \text{positive small}, \\ &\mathrm{PB} = \text{positive big}\}\end{aligned}$$

The following set of rules describes the behaviour of the controller

$$\begin{array}{ll}\textbf{if } x \textbf{ is } \mathrm{EC} \textbf{ then } y = \mathrm{PB}, & \textbf{if } x \textbf{ is } \mathrm{W} \textbf{ then } y = \mathrm{NS}, \\ \textbf{if } x \textbf{ is } \mathrm{VC} \textbf{ then } y = \mathrm{PS}, & \textbf{if } x \textbf{ is } \mathrm{H} \textbf{ then } y = \mathrm{NS}, \\ \textbf{if } x \textbf{ is } \mathrm{C} \textbf{ then } y = \mathrm{PS}, & \textbf{if } x \textbf{ is } \mathrm{VH} \textbf{ then } y = \mathrm{NB}, \\ \textbf{if } x \textbf{ is } \mathrm{M} \textbf{ then } y = \mathrm{ZO}. & \end{array}$$

This rule base leads to the following relation R written as a matrix with rows corresponding to $\mathrm{LV}_{\mathrm{in}}$ and columns corresponding to $\mathrm{LV}_{\mathrm{out}}$.

$$R = \begin{pmatrix} 0&0&0&0&1 \\ 0&0&0&1&0 \\ 0&0&0&1&0 \\ 0&0&1&0&0 \\ 0&1&0&0&0 \\ 0&1&0&0&0 \\ 1&0&0&0&0 \end{pmatrix}$$

The fuzzy sets Q_i and S_j corresponding to the linguistic variables may be choosen as usual by some convenient membership relation. As a defuzzification we take $\mathcal{D}(X) = \mathrm{glb}_E(((X;\mathbb{T}\sqcap\mathbb{I})\backslash X)^{\downarrow})$, where E is the ordering on $[-20,20]$. The cut within $\mathcal{D}$ selects the maximal values in every row of X. Afterwards, in every row we choose the greatest lower bound in respect to E, such that we get a function and hence for every input temperature exactly one output value.

3 Optimization Problems

The underlying lattice $\mathcal{L}$ may be interpreted as a partial order of degrees of optimality for a problem. A vector $V : 1 \to A$, i.e.,a relation from a singleton set to set A, may describe all possible solutions of the problem. If V is an $\mathcal{L}$-fuzzy relation it also describes the degree of optimality of every solution. As in the last section we may take a suitable cut of V to get an optimal solution.

$$\text{The problem has an optimal solution} \iff ((V;\mathbb{T}_{A1})\backslash V)^{\downarrow} \neq \perp\!\!\!\perp_{1A}.$$

Notice, that we take a cut with $V;\mathbb{T}_{A1}$ instead of $V;\mathbb{T}_{A1}\sqcap\mathbb{I}_1$ since $\mathbb{I}_1 = \mathbb{T}_{11}$. If the problem has an optimal solution we get the following statements:

1. Any crisp point x, i.e., x vector, $x^{\uparrow} = x, x \neq \perp\!\!\!\perp_{1A}$ and $x^{\smile};x \sqsubseteq \mathbb{I}_A$, with $x \sqsubseteq ((V;\mathbb{T}_{A1})\backslash V)^{\downarrow}$ is an optimal solution.
2. The degree of an optimal solution is given by the scalar $V;\mathbb{T}_{A1}$.

As an example we want to describe a common situation in a shopping mall. One has to select a set of products which fulfil some optimality criteria. In our example this would be

$\mathrm{c} \triangleq$ Product is cheap.	$\mathrm{t} \triangleq$ Product tastes very well.
$\mathrm{q} \triangleq$ Product has a great quality.	$\mathrm{a} \triangleq$ Product is accepted by the family.

As the lattice $\mathcal{L}$ we take the Boolean lattice with atoms $\{\mathrm{c},\mathrm{q},\mathrm{t},\mathrm{a}\}$. Suppose P is the following $\mathcal{L}$-fuzzy vector of products $\{P_1, P_2, P_3, P_4\}$ with their properties.

$$P = \begin{array}{cccc} P_1 & P_2 & P_3 & P_4 \\ (\{\mathrm{q},\mathrm{t}\} & \{\mathrm{c},\mathrm{q},\mathrm{t}\} & \{\mathrm{q},\mathrm{t},\mathrm{a}\} & \{\mathrm{c},\mathrm{q}\}) \end{array}$$

Let ε_0 the crisp membership relation of nonempty sets of products, i.e., $\varepsilon_0(P_i, M)$ holds iff P_i is an element of the nonempty set M of products. This relation may be visualized by the matrix with rows corresponding to the set $\{P_1, P_2, P_3, P_4\}$ and the columns corresponding to the nonempty sets of products

$$\varepsilon_0 = \begin{pmatrix} 0&0&0&0&0&0&0&1&1&1&1&1&1&1&1 \\ 0&0&0&1&1&1&1&0&0&0&0&1&1&1&1 \\ 0&1&1&0&0&1&1&0&0&1&1&0&0&1&1 \\ 1&0&1&0&1&0&1&0&1&0&1&0&1&0&1 \end{pmatrix}$$

where 1 is a shorthand for $\{\mathrm{c},\mathrm{q},\mathrm{t},\mathrm{a}\}$ and 0 for $\emptyset$. Furthermore, suppose f is the following subobject of legal sets of products (a crisp injective mapping), i.e., the subobjects of those sets of products which we are able to buy.

$$f = \begin{pmatrix} 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \end{pmatrix}$$

f expresses the fact that we are not allowed to buy just one product and P_2 and P_4 together. Now, the $\mathcal{L}$-fuzzy vector V of possible solutions is given by the term $\mathrm{syQ}(P^\smile, \varepsilon_0); (\varepsilon_0 \backslash \varepsilon_0)^\smile; f^\smile$ and as matrix by

$$\begin{array}{ccccccc} \{P_3,P_4\} & \{P_2,P_3\} & \{P_1,P_4\} & \{P_1,P_3\} & \{P_1,P_3,P_4\} & \{P_1,P_2\} & \{P_1,P_2,P_3\} \\ \end{array}$$
$$\left(\begin{array}{ccccccc} \{\mathrm{q}\} & \{\mathrm{q},\mathrm{t}\} & \{\mathrm{q}\} & \{\mathrm{q},\mathrm{t}\} & \{\mathrm{q}\} & \{\mathrm{q},\mathrm{t}\} & \{\mathrm{q},\mathrm{t}\} \end{array} \right)$$

Therefore, the optimal solutions $((V;\pi_{A1})\backslash V)^\downarrow$ are characterized by the (crisp) vector $\left(\begin{smallmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{smallmatrix}\right)$ corresponding to the product sets $\{P_2,P_3\},\{P_1,P_3\},\{P_1,P_2\}$ and $\{P_1,P_2,P_3\}$. The degree of such an optimal solution may be computed as $V;\pi_{A1}$ corresponding to the lattice element $\{\mathrm{q},\mathrm{t}\}$.

4 Games

As shown in [8] a strategy of a game corresponds to the kernel of the graph belonging to the game. Such a kernel can be computed by relational methods (cf. [8]).

We want to study a variant of the well-kown two person NIM game. The rules of this game are as follows. At the beginning there is a fixed number of matches on a table (in our expample 21). Every player is allowed to remove 1 or 2 matches. There are two special moves. Once in the game one player is allowed to remove 3 matches and once in the game one player is allowed to remove 4 matches. If there is no match left the player who is on move loses.

We model this game using the Boolean lattice $\mathcal{L}$ with atoms $\{3,4\}$ for the special moves. An element of $\mathcal{L}$ describes which special move is still available. The graph of the game is a relation R between the number of matches left on the table. It is defined by

$$\begin{array}{ll} R(n,m) = \emptyset & \text{iff } n-m \geq 5 \text{ or } n=m, \\ R(n,m) = \{4\} & \text{iff } n-m=3, \\ R(n,m) = \{3\} & \text{iff } n-m=4, \\ R(n,m) = \{3,4\} & \text{iff } n-m \in \{1,2\}, \end{array}$$

corresponding to the interpretation of $\mathcal{L}$ indicated above. The kernel of this relation is $\mathcal{L}$-fuzzy vector on the set $\{0,\dots,21\}$ of the number of matches left on the table. Its matrix representation looks like

$$\begin{array}{cccccccccccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 \end{array}$$
$$\left(\begin{array}{cccccccccccccccccccccc} 1 & 0 & 0 & 3 & 4 & 0 & 3 & 0 & 4 & 3 & 0 & 0 & 1 & 0 & 0 & 3 & 4 & 0 & 3 & 0 & 4 & 3 \end{array} \right)$$

where $0,3,4,1$ are shorthands for $\emptyset,\{3\},\{4\},\{3,4\}$, respectively. This kernel may be interpreted as follows. If at least the special move 3 is left state 9 is a winning state for the player who is on move. He takes 3 matches and the game is in state 6. The other player may switch to state $5,4$ or to state 2 by using the special move 4. In the second case the first player can go immediately to state 0 and win, in the first case he can go to state 3 such that the second player just may switch to state 1 or 2 where he will lose agin. Analogously, state 12 is a winning state for the player who is on move if both special moves are still available, and state 10 is a winning state if both special moves are not available. The kernel above shows that the player who starts will win the game. But he will lose if the special move 3 is forbidden. In this case he may switch to state $20,19$ or 17. In all cases the second player can go to a state which is labelled by 3 (state 15 or 18).

References

1. Chin L.H., Tarski A.: Distributive and modular laws in the arithmetic of relation algebras. University of California Press, Berkley and Los Angeles (1951)
2. Freyd P., Scedrov A.: Categories, Allegories. North-Holland (1990).
3. Goguen J.A.: L-fuzzy sets. J. Math. Anal. Appl. 18 (1967), 145-157.
4. Jónsson B., Tarski A.: Boolean algebras with operators, I, II, Amer. J. Math. 73 (1951) 891-939, 74 (1952) 127-162
5. Kawahara, Y., Furusawa H.: Crispness and Representation Theorems in Dedekind Categories. DOI-TR 143, Kyushu University (1997).
6. Mamdani, E.H., Gaines, B.R.: Fuzzy Reasoning and its Application. Academic Press, London (1987)
7. Olivier J.P., Serrato D.: Catégories de Dedekind. Morphismes dans les Catégories de Schröder. C.R. Acad. Sci. Paris 290 (1980) 939-941.
8. Schmidt G., Ströhlein T.: Relationen und Graphen. Springer (1989); English version: Relations and Graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoret. Comput. Sci., Springer (1993)
9. Schmidt G., Hattensperger C., Winter M.: Heterogeneous Relation Algebras. *In:* Brink C., Kahl W., Schmidt G. (eds.), Relational Methods in Computer Science, Advances in Computer Science, Springer Vienna (1997).
10. Winter M.: Strukturtheorie heterogener Relationenalgebren mit Anwendung auf Nichtdetermismus in Programmiersprachen. Dissertationsverlag NG Kopierladen GmbH, München (1998)
11. Winter M.: A new Algebraic Approach to L-Fuzzy Relations Convenient to Study Crispness. INS Information Science, in print
12. Zadeh L.A.: Fuzzy sets, Information and Control 8 (1965), 338-353.

Internet-based Experimentation with Heterogeneous Software Tools

Volker Braun

Universität Dortmund, Fachbereich Informatik, D-44221 Dortmund, Germany

Abstract This paper gives an overview of the Electronic Tool Integration (ETI) platform (see also [9] and [2]), a platform for the interactive experimentation with heterogeneous software tools via the Internet. This platform enables even newcomers to master the wealth of existing software tools in a short timespan, and to identify the most appropriate collection of tools to solve their own application-specific tasks.

1 The Goals

Modern software engineering is more and more dependent on automation and good tool support. However, faced with a problem, it is hard to identify the appropriate tools. In particular, since generic tools are often not adequate, (different) tools having a specific focus are needed to tackle the task. Of course, the Internet is a good resource for information. But the advantage of the available information-variety is extenuated by the fact that the right software tool is difficult to find. Though, there is generic support for each step of the tool evaluation process, in form of

1. searching the Web,
2. reading the available documentation,
3. installing the software tool and finally
4. experimenting with the tool, tool-evaluation

specific support and overlapping assistance is still missing.

The Electronic Tool Integration (ETI) platform overcomes this problem by providing Internet-based access to software tools with the ability to

- retrieve information on each available tool,
- execute single tool features, and
- combine features coming from different tools within the repository and run the corresponding programs.

These features are offered to the public via moderated Internet sites, called *ETI Sites*, which give access to a collection of pre-installed software tools using the ETI platform.

This paper presents the user's view of the ETI platform in form of the *ToolZone Software*. For this, we first give a short overview to this Internet-based client/server application in Sect. 2. This introduces terms like *Activities* (see Sect. 3), *Taxonomies* (see Sect. 4) and *Coordination Programs* (see Sect. 5 and Sect. 6) which are afterwards explained in the subsequent sections.

2 Introducing the ToolZone Software

A user can experiment with the tools hosted by an ETI site using the *ToolZone Software*. This client/server application gives access to a tool repository in which tool features are represented as functional entities called *ETI Activities* (see Sect. 3). Additionally, the tool repository comprises the data types the activities work on. The activities and the associated data types are classified for ease of retrieval according to behavioral and interfacing criteria via ETI's *Type and Activity Taxonomy*, respectively (see Sect. 4). Using the ToolZone software, the end user can get information on each activity and type contained in the tool repository via ETI's taxonomy browsers (see Fig. 1) beside others, in form of a link to the underlying tool's home page or contact address. Whereas links to tools having a specific focus are also available via other Web sites like the Petri Nets Tool Database [3] or the Formal Methods Europe [1] database, the key feature of the ToolZone software is its unique Internet-based experimentation facility. This means, that via the ToolZone software end users can execute single activities as well as combine activities to coordination programs and finally execute the programs via the Internet.

To combine (coordinate) the activities contained in the tool repository, experienced users can make use of ETI's procedural coordination language *HLL (High-Level Language)* [6] to manually implement the intended coordination task on the basis of the available activities (see Sect. 5). Unexperienced users are supported by means of *ETI's Synthesis Component* [10] which generates sequences of activities out of abstract specifications (see Sect. 6). Single activities or combinations of them can be run via the Internet on libraries of examples, case studies, and benchmarks which are also available in the tool repository. Additionally, the user can experiment with own sets of data, to be deployed in user-specific, protected home areas.

3 Activities

In general, tools are not integrated into the tool repository as monolithic blocks. Rather, single tool features are identified and prepared to be accessible by the platform. Within the tool repository, a distinct tool feature is represented by an ETI Activity which is the elementary functional component of the ETI platform. An activity definition comprises

- the *name* that is beside others required to reference the activity in a coordination-task description (see Sect. 6).
- the *classification constituent* in terms of predicates which characterizes the activity within the tool repository. This information can be used to specify an activity using abstract properties instead of its name in a coordination-task description.
- the *implementation constituent* which implements the activity on the basis of the corresponding tool feature.

After an activity has been made available within the tool repository, it can be combined with other activities. For this, the end user can write an HLL-program implementing the intended coordination task. Alternatively, the glue-code coordinating the activities can be generated automatically on the basis of an abstract coordination-task description, called *Loose Specification* (see Sect. 6).

In contrast to HLL-based coordination which can be used to combine arbitrary activities contained in the tool repository, only activities having a certain profile can be used for program synthesis. These *synthesis-compliant activities* look at a tool feature as a "transformational" entity. This means that a tool feature is seen as a component taking an object of type T_1 as input and delivering an object of type T_2 as output.

As an example, Table 1 shows the names and the interface constituent of the definition of synthesis-compliant activities identified within the CADP toolkit [5]. It presents the name of the activities, their input and output types as well as the tools their implementation is based on.

Activity Name	Input Type	Output Type	Tool	Description
openAUTFile	ETINone	AUTFile	aldebaran	Loads an AUT-File object.
aldebaran_MIN_STD_I	AUTFile	AUTFile	aldebaran	Minimizes an LTS.
autF2bcgF	AUTFile	BCGFile	aldebaran	Transforms a file in the aut format into the BCG file format.
bcgEVAL	BCGFile	ETIFile	bcg_open	A model checker based on bcg_open.
xtl	BCGFile	ETIFile	xtl	Evaluation of value-based temporal logic formulas.

Table 1. Activities Based on the CADP Toolkit

4 Taxonomies

For a flexible handling (retrieval, loose object specification, abstract views), activities and the data types they work on are classified by means of the *Activity Taxonomy* and *Type Taxonomy*, respectively. A taxonomy is a hierarchical structure of predicates over a set of atomic elements, here the activities and types respectively. Formally, a taxonomy (defined over a set of atomic objects S) is a sub-lattice of the power-set lattice over S which comprises elements with a particular profile which are identified by names. Taxonomies can be represented as directed acyclic graphs (DAGs) where each leaf represents an atomic entity (here activity or type) and each intermediate node represents a set of entities, called group. Conceptually, edges reflect an "is-a" relation between their target and source nodes. The semantics of an intermediate node is the set of all atomic entities which are reachable within the taxonomy DAG from this node. With respect to this semantics, edges reflect the standard set inclusion.

Fig. 1 shows a simple classification of the types introduced in Table 1 within ETI's taxonomy browser. Here, the type group `CADPFile` represents beside others the types `LOTOSFile`, `SEQFile`, `BCGFile`, `AUTFile`, and `EXPFile`. The activities are organized within the activity taxonomy analogously.

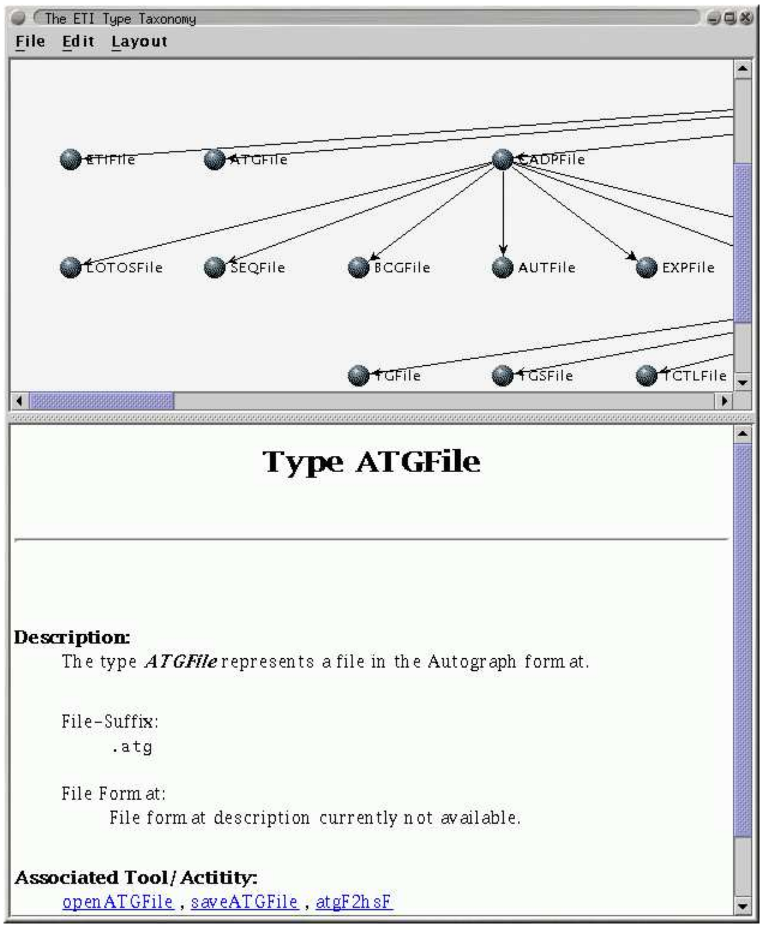


Figure 1. A simple Type Taxonomy

5 HLL Programming

Once an activity is available in the tool repository, it can be accessed via the HLL (High-Level Language) function defined by the activity's implementation constituent (see Sect. 3). On the basis of this HLL function, HLL programs are used to manually combine activities representing heterogeneous functionalities coming from different tools in order to perform complex tasks.

The HLL-based coordination program presented in Fig. 2 minimizes a labelled transition system (LTS) stored in the aut-format, a file format defined by the Caesar/Aldebaran Development Package (CADP). After that, the `xtl` model checker, a tool contained in the CADP toolkit, is invoked on the minimized labelled transition system. In detail, this is done by requesting from the user the files storing the labelled transitions system and the formula to be checked via the `fsBoxLoad` function of the ETI HLL-library (see (1) and (2) of Fig. 2). Then the model is minimized with respect to observational equivalence [8] using the `aldebaranMIN_STD_I` function contained in the HLL-library `CADP` (see (3)). Since the `xtl` model checker requires the model to be provided in the bcg file format, the aut-representation of the minimized labelled transition system is transformed into this format by the HLL function `autF2bcgF` (see (4)). Finally, the `xtl` model checker is called using the `xtl` function of the HLL-library `CADP` getting the model and the formula as arguments (see (5)).

```
var String: aut_model;
var String: min_aut_model;
var String: bcg_model;
var String: formula;
var ETIResult: result;
aut_model := ETI.fsBoxLoad ("Select Model", "*.aut"); (1)
formula := ETI.fsBoxLoad ("Select Formula", "*.xtl"); (2)
result := CADP.aldebaranMIN_STD_I (aut_model, min_aut_model); (3)
result := CADP.autF2bcgF (min_aut_model, bcg_model); (4)
result := CADP.xtl (bcg_model, formula); (5)
```

Figure 2. A HLL-based Coordination Program

6 Program Synthesis

In addition to the HLL-based coordination, the ETI platform provides automated coordination support by means of its synthesis component. Here, the glue-code combining the activities is automatically generated. For this, the user specifies a coordination task via an abstract description called *Loose Specification* (see Fig. 3 as an example), instead of programming it using the HLL. Using loose specifications, the user characterizes what he wants to achieve instead of how to achieve it. This goal-oriented approach is the main difference between ETI's synthesis-based coordination facility and other coordination approaches like UNIX piped commands, scripting languages (e.g. Perl [11], Python [7]) or the ToolBus [4]: there the user is forced to precisely specify the coordination process like in HLL programs.

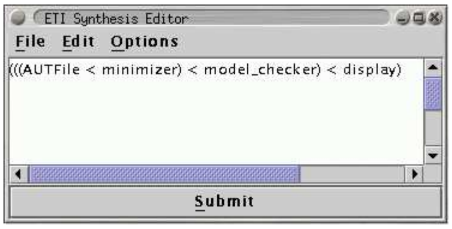


Figure 3. A Loose Specification

These abstract descriptions which are based on the Semantic Linear-time Temporal Logic [10] are loose in two orthogonal dimensions:

Local Looseness : The characterization of types and activities is done at the abstract level of the taxonomies, instead of enumerating them explicitly. Here names contained in the taxonomies are interpreted as propositional predicates (see e.g. the activity groups `minimizer`, `model_checker`, and `display` in Fig. 3). They can be combined by the Boolean operators & (and), | (or) and ~ (not) to specify sets of activities and types.

Global/Temporal Looseness : The characterization of whole coordination sequences is done in terms of abstract constraints specifying precedences, eventuality, and conditional occurrence of single taxonomy entities, rather than specifying the precise occurrence of the types and activities (see e.g. the before operator $<$ in Fig. 3).

From the coordination-task description, the synthesis component then generates sequences of activities (called *Coordination Sequences*) each implementing the specified task. Coordination sequences are finite paths of the form

$$T_1 \xrightarrow{a_1} T_2 \xrightarrow{a_2} T_3 \cdots T_{n-1} \xrightarrow{a_{n-1}} T_n$$

where T_i is a type and a_i is a synthesis-compliant activity which transforms an object of type T_i into an object of T_{i+1}.

As result of the synthesis process all coordination sequences which satisfy a given loose specification are presented to the user as directed graph called the *Synthesis Solution Graph* (see Fig. 4).

Using the ToolZone software, the user can graphically select his favored sequence within the synthesis solution graph and then run the corresponding program via the Internet.

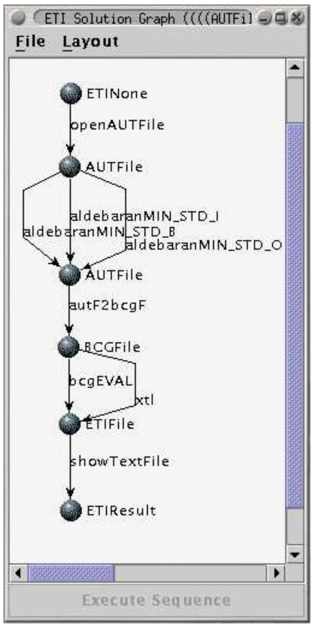


Figure 4. A Synthesis Solution Graph

7 Conclusion

Within this paper we have presented the user's view of the Electronic Tool Integration platform which allows the experimentation with heterogeneous software tools via the

Internet. On the basis of a set of activities, which represent distinct features of pre-installed software tools, the user can build coordination programs manually using the HLL. Additionally, sequential compositions of activities can be generated out of goal-oriented abstract specifications. Coordination programs can then be run via the Internet.

We are optimistic that this platform will help overcoming the typical hesitation to try out new technologies: serious hurdles, like installation of the tools, getting acquainted with new user interfaces, lack of direct comparability of the results and of performances, are eliminated.

8 Acknowledgements

I am grateful to Bernhard Steffen and Tiziana Margaria for the years' long cooperation on the ETI platform and to the whole ETI team, in particular Andreas Holzmann, for their support.

References

1. Formal Methods Europe. http://www.fmeurope.org/.
2. The ETI Community Online Service. http://www.eti-service.org/.
3. The Petri Nets Tool Database. http://www.daimi.aau.dk/PetriNets/tools/db.html.
4. J. Bergstra and P. Klint. The ToolBus coordination architecture. In *Coordination Languages and Models (COORDINATION '96)*, volume 1061 of *Lecture Notes in Computer Science (LNCS)*, pages 75–88, Heidelberg, Germany, 1996. Springer-Verlag.
5. J. C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. Cadp: A protocol validation and verification toolbox. In T. A. Henzinger R. Alur, editor, *Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science (LNCS)*, Heidelberg, Germany, 1996. Springer-Verlag.
6. A. Holzmann. Der METAFrame-Interpreter: Entwicklung und Implementierung eines dynamischen Modulkonzeptes, 1997.
7. M. Lutz. *Programming Python*. O'Reilly & Associates, 2001.
8. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
9. B. Steffen, V. Braun, and T. Margaria. The Electronic Tool Integration Platform: Concepts and Design. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 1(1/2):9–30, 1997. Springer-Verlag, Heidelberg, Germany.
10. B. Steffen, T. Margaria, and B. Freitag. Module configuration by minimal model construction. In *Proc. Workshop on Semantikgestützte Analyse, Entwicklung und Generierung von Programmen, GI-Fachgruppe 2.1.3, Schloß Rauischholzhausen (Germany)*, number 94-02 in Technical Report. Justus-Liebig-Universität Giessen, March 10-11 1994.
11. L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, 3rd edition edition, 2000.

TEMPLUS — TEaching Management PLatform for UniversitieS

Claudia Gsottberger

Lehrstuhl Informatik V
Universität Dortmund
Claudia.Gsottberger@cs.uni-dortmund.de

Zusammenfassung Im Rahmen von TEMPLUS entsteht eine internetbasierte, personalisierte Plattform, die es den verschiedenen universitären Nutzergruppen ermöglichen soll, effizient zu kooperieren. Dies beinhaltet insbesondere eine uniforme Integration der angebotenen Dienstleistungen (kurz: Dienste) in einer Form, die eine rollen- und profilspezifische Zugangskontrolle und Nutzerführung erlaubt.

1 Motivation

Universitärer Lehrbetrieb beinhaltet Organisation und Durchführung von Übungen. Die Organisation umfasst die Berücksichtigung verschiedener universitärer Übungsformen, die Anmeldung der Studierenden zu den Übungen, die Verteilung der Studierenden auf die einzelnen Übungsgruppen, die Schulung der Tutoren, sowie das Erstellen von Statistiken. Bei der Durchführung von Übungen sind zentrale Punkte das Erstellen und Verteilen von Übungsblättern und Musterlösungen, die Korrektur von Abgaben, die Präsentation von Lerninhalten und -materialien, sowie die Feedback-Erfassung.
Dabei sind hohe Studierendenzahlen, stark wachsende Studienzeiten, hohe Abbruchquoten, eine unzureichende Betreuungssituation, hohe Belastung des Lehrpersonals und komplexe, sich häufig ändernde Workflows unsere täglichen Herausforderungen.
An der Universität Dortmund wurde aufgrund dieser Problematik ein WIS-Projekt (Sofortprogramm zur Weiterentwicklung des Informatikstudiums) ins Leben gerufen. Mehrere Lehrstühle des Fachbereichs Informatik arbeiten im Rahmen dieses Projektes an der Verbesserung der Lehre im Grundstudium. Eines der Teilprojekte ist TEMPLUS.

2 Ziele

Die Ziele des TEMPLUS–Projekts sind die Entlastung des Lehrpersonals von administrativen Aufgaben und eine Verbesserung der Betreuung der Studierenden. Dies soll erreicht werden durch eine proaktive Workflow-Steuerung, die Ergänzung der Kommunikation durch elektronische Medien, sowie das automatische Erfassen und Auswerten statistischer Informationen.
Anforderungen an das zu entwickelnde System und unser Lösungsansatz dafür sind:

- **Flexibilität**: Änderungen in den komplexen Workflows müssen einfach und schnell im System umzusetzen sein. Eine komponentenbasierte, graphische Konfiguration der Workflows bietet dabei viele Vorteile.

- **hohe Verfügbarkeit**: Viele Nutzer arbeiten von unterschiedlichen Orten und unterschiedlichen Betriebssystemen aus mit dem System. Es bietet sich daher an, eine webbasierte Plattform zu entwickeln.
- **Personalisierbarkeit**: Die verschiedenen Nutzergruppen (z.B. Studenten, Dozenten, Tutoren, etc.) haben unterschiedliche Rechte und Pflichten in Bezug auf Dateneinsicht und -änderung. Ein flexibles Nutzer- und Rollenmanagement bietet die notwendige Kontrolle.
- **Zuverlässigkeit**: Sicherheitskritische Workflows erfordern die Korrektheit der Abläufe. Es ist also notwendig, die entworfenen Workflows in Bezug auf bestimmte Kriterien zu validieren.

3 Technologie

Das *Agent Building Center* (*ABC*) ist eine generische graphische Entwicklungsumgebung für anwendungsspezifisches, komponentenbasiertes Softwaredesign [5]. Dabei wird der Workflow als Graph (*Service Logic Graph*, kurz *SLG*) modelliert. Dieser besteht aus Knoten, den funktionalen Einheiten (sog. *SIBs*, d.h. *Service Independent Building Blocks*), und Kanten, die den Kontrollfluss repräsentieren (vgl. Abb. 1). Die Trennung der Daten (Implementierung der *SIBs*) von dem konkreten Work-

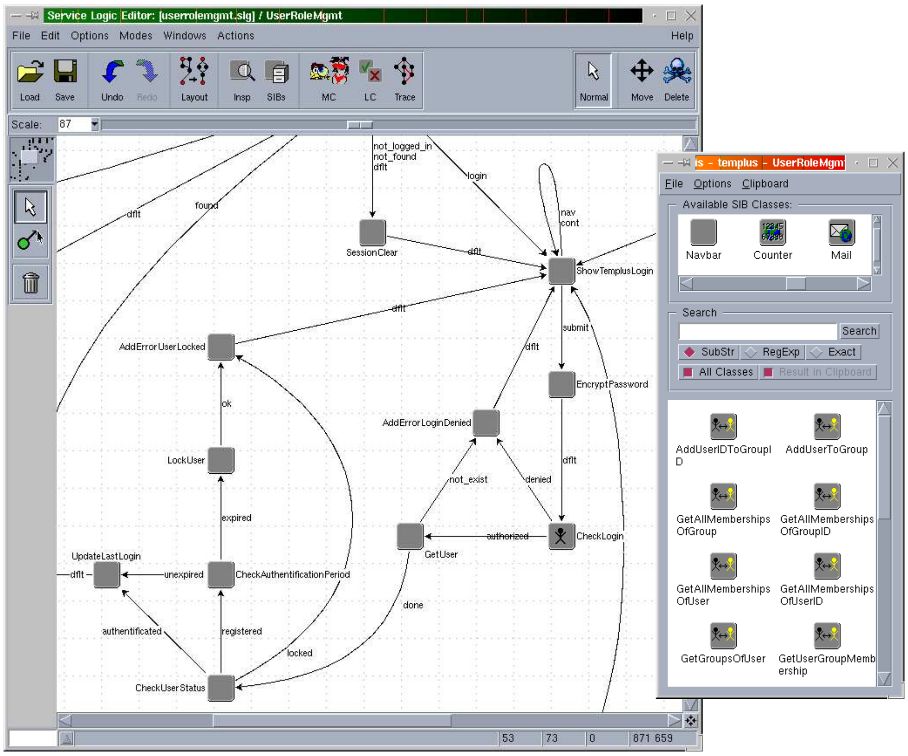


Abbildung 1. Graphische Konfiguration von Workflows mit ABC

flow (*Service Logic Graph*) ermöglicht Aufgabenteilung und Spezialisierung während

der Dienstentwicklung. Der zugehörige Software-Entwicklungsprozess ist in [1,3] beschrieben.
Die graphische Konfiguration der Workflows in Form der *SLG*s liefert die nötige Flexibilität, um die Workflows an Änderungen anzupassen.
Zusätzliche Module innerhalb des ABC garantieren die Korrektheit und Zuverlässigkeit der entwickelten Dienste und vereinfachen die Dienst-Entwicklung:

- Workflow-Validierung in Form von Überprüfung lokaler und globaler Constraints, sowie symbolische Ausführung [2,6].
- Automatische Generierung der Applikation, d.h. der Übergang von der Spezifikation (*SLG*) zur Implementierung (konkreter Dienst) wird durch einen Compile-Schritt realisiert.
- Einsatz vorgefertigter Komponenten (*SIB*s) für bestimmte Aufgabenbereiche, z.B. Nutzer- und Rollenmanagement [4]
- Makros machen komplexe Workflows beherrschbar und unterstützen die Aufgabenteilung während der Dienst-Entwicklung (vgl. Abb. 2).

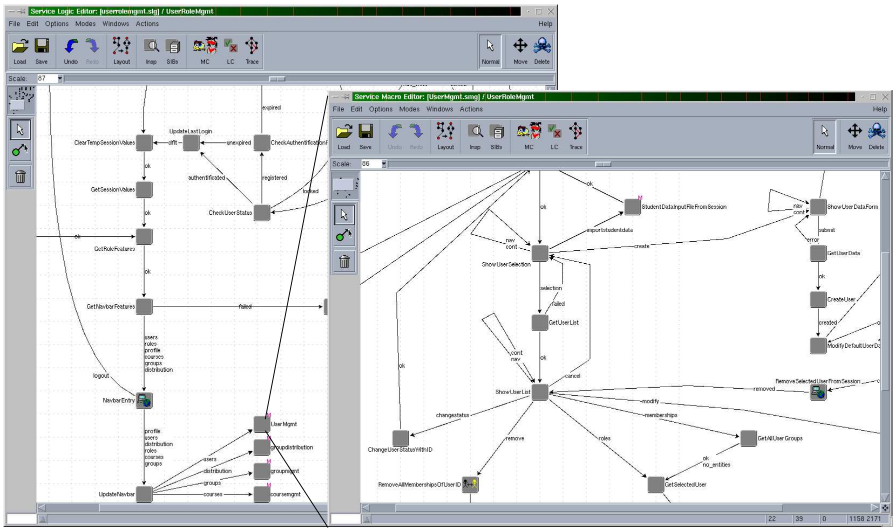


Abbildung 2. Makros im ABC

4 Der TEMPLUS–Dienst

Der TEMPLUS-Dienst ist ein personalisierter, webbasierter Internetdienst für die Organisation und Durchführung von Lehrveranstaltungen an der Universität.
Folgende Teilfunktionalitäten sind zur Zeit im TEMPLUS-Dienst umgesetzt:

- Flexibles Nutzer- und Rollenmanagement angepasst an die Bedürfnisse und Rechte der universitären Nutzergruppen
- Einsatz von Personalisierung als Filtermechanismus für Daten und Funktionalitäten
- Verwaltung von Vorlesungen und Übungen

- Anmeldung zu Vorlesungen und Übungen
- Automatische Verteilung auf Übungsgruppen, sowie die Möglichkeit, per Hand Studierende nachträglich einzuteilen bzw. zu verschieben.
- Importfilter für Studierendendaten zum Einrichten von Default-Accounts für den TEMPLUS-Dienst

Zentral für alle Teilbereiche des TEMPLUS-Dienstes sind **Personalisierung** und **Proaktivität**.

Die **Personalisierung** erlaubt eine adäquate Behandlung von Rollen im universitären Alltag (z.B. Student, Dozent, Tutor, etc.). Zugang zum Dienst und damit zu den darüber angebotenen Funktionalitäten und erreichbaren Daten erfolgt nur mittels eines persönlichen Logins und Passworts. Nachdem sich ein Benutzer eingeloggt hat, werden ihm abhängig von seiner Rolle verschiedene Funktionalitäten angeboten (vgl. Abb.3). Die persönliche Navigationsleiste zeigt den Benutzernamen, die Rollen des Benutzers, sowie seine aktive Rolle und je einen Button für eine Klasse von Funktionalitäten, die für diese Rolle erlaubt sind.

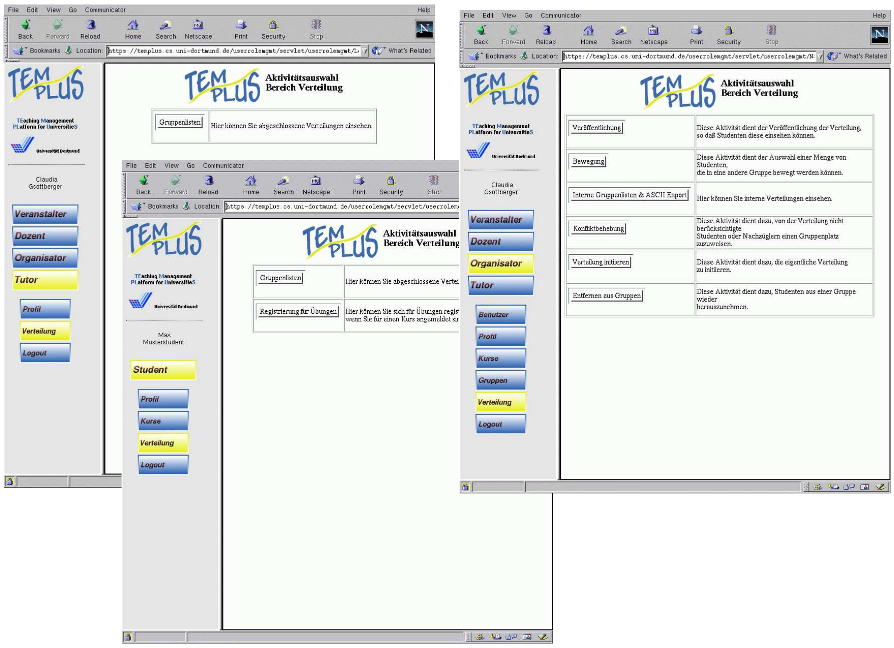


Abbildung 3. Rollenspezifisches Anbieten von Funktionalitäten

Durch die **Proaktivität** des TEMPLUS-Dienstes werden darüberhinaus, rollen- und zeitspezifisch Funktionalitäten aktiviert oder deaktiviert bzw. unterscheiden sich im Ablauf.
Der Button *Verteilung* führt z.B. zu allen Aktivitäten, die mit der Einteilung von Übungsgruppen zu tun haben. Dabei werden die Aktivitäten rollenabhängig aktiviert bzw. deaktiviert (vgl. Abb.3).

Aktivitäten können je nach Rolle unterschiedlich ablaufen, d.h. unterschiedliche Sichten auf die darunterliegende Datenmenge bieten. Beispielsweise hat der Organisator die Teilnehmerlisten der Übungsgruppen inklusive der persönlichen Daten der Studierenden, wie z.B. Studiengang, Fachsemester, Matrikelnummer, während die Tutoren nur die Namen der Teilnehmer in ihren Übungsgruppen kennen, und die Studierenden nur wissen müssen, für welche Übungsgruppe sie selbst eingeteilt sind.
Die zeitspezifische Aktivierung bzw. Deaktivierung von Funktionalitäten ist applikationsabhängig. Im Rahmen der Anmeldung zu Übungsgruppen ist ein Anmeldezeitraum für eine Veranstaltung definiert. Studierende dürfen das Anmeldeformular nur während dieses Zeitraums editieren. Nach Ablauf der Anmeldefrist können Studierende lediglich ihre eingegebenen Daten einsehen, der Organisator kann jedoch erst danach den Verteilungsalgorithmus anstoßen, sowie dessen Ergebnisse veröffentlichen, nachbessern und Konflikte beheben. Studierende und Tutoren können die Ergebnisse erst nach der Veröffentlichung einsehen.

5 Zusammenfassung und Ausblick

Der TEMPLUS-Dienst ist ein personalisierter, webbasierter Internetdienst zur Organisation und Durchführung von Lehrveranstaltungen an der Universität. Zentral dabei sind die adäquate Handhabbarkeit der Rollen des universitären Alltags, sowie die Zuverlässigkeit der entwickelten Workflows, um die Datensicherheit im Rahmen der sicherheitskritischen Abläufe zu gewährleisten.
Das Agent Building Center (*ABC*) [5] ist die ideale Entwicklungsumgebung für eine derartige Webapplikation, denn es bietet

- Komponentenbasierte, graphische Entwicklung workflow-zentrierter Software: Somit ist Flexibilität in Design, Wartung und Weiterentwicklung gewährleistet.
- Flexibles Nutzer- und Rollenmanagement in Form einer Bibliothek von Komponenten: Es ist z.B. auch möglich, zur Laufzeit neue Nutzer einzurichten und Rollen zu definieren bzw. zu modifizieren.
- Workflow-Validierung auf dem Ablaufniveau (*SLG*): Dabei werden die Komponenten (*SIB*s) als korrekt funktionierende atomare Bausteine betrachtet. Die Validierung der Workflows ist der Schlüssel zu zuverlässigen Webapplikationen. Symbolische Ausführung der Applikation, lokale Checks bzgl. der Konfiguration einzelner *SIB*s und globale Checks, d.h. Überprüfen der Interaktion zwischen verschiedenen *SIB*s innerhalb des Workflows mittels Modelchecking, ermöglichen es dem Benutzer zur Designzeit (d.h. bevor die konkrete Applikation generiert und getestet wird) die Konsistenz seiner Applikation prüfen. So werden viele Fehler bereits in einer sehr frühen Phase des Softwareentwicklungsprozesses gefunden, was die Kosten und die Entwicklungszeit der Webapplikationen reduziert.

Unmittelbare Feuerprobe für den Dienst ist der Einsatz bei der Organisation und Durchführung der Erstsemestervorlesung *Datenstrukturen, Algorithmen und Programmierung 1* im WS 2001/02 an der Universität Dortmund. An dieser Stelle möchte ich Bernhard Steffen und Volker Braun für ihre inhaltlichen Beiträge, dem TEMPLUS-Entwicklerteam für die gute Zusammenarbeit, sowie dem METAFrame-Team für die technische Unterstützung danken. Ohne ihre Mitarbeit wäre es nicht möglich gewesen, den TEMPLUS-Dienst innerhalb weniger Monate zu entwickeln.
Geplante Erweiterungen für TEMPLUS, die zum Teil noch während des WS 2001/02 zum Einsatz kommen sollen, sind

- eine Statistik-Komponente
  - zur Verwaltung der Anwesenheit in den Übungen und den bei der Abgabe von Lösungen zu Übungsaufgaben erreichten Punkten
  - mit automatischer Überprüfung von Scheinkriterien
- ein Nutzerprofil-Generator für Studierende, um den Wissensstand und den Lernfortschritt bewerten zu können, sowie die Ursachen für Probleme benennen zu können
- Verwaltung von Übungsaufgaben
- Online-Abgabe und Korrekturunterstützung (z.B. automatische Compilierung bei Programmieraufgaben als Präprozess)
- profilabhängige Aufbereitung von Lerninhalten bzw. Generierung persönlicher Lerneinheiten zur Nachbereitung der Vorlesungsinhalte bzw. zur Prüfungsvorbereitung

bis hin zu

- Notebook-Klausur, bei der alle Studierenden ihre persönlichen, parametrisierten Aufgaben bekommen [7]
- flexiblem Übungsbetrieb mit themenspezifischen Übungsgruppen und Schwerpunkttutorien

Literatur

1. V. Braun, A. Holzmann, S. Bollin, and K. Plociennik. *The Agent Building Center: Bringing a Web Application into Operation.* METAFrame Technologies GmbH, Dortmund, Germany, 2001.
2. V. Braun, T. Margaria, and B. Steffen. Personalized electronic commerce services. In *Proc. ICTS'2000, 8th Int. Conference on Telecommunication Systems Modeling and Analysis, March 9-12, 2000, Nashville, Tennessee, USA*, 2000. to appear also in the Telecommunication Systems Journal.
3. Volker Braun. *A Coarse-granular Approach to Software Development allowing Non-Programmers to Build and Deploy Reliable, Web-based Applications.* PhD thesis, University of Dortmund, Dortmund, Germany. to appear.
4. B. Lindner, T. Margaria, and B. Steffen. Ein personalisierter internetdienst für wissenschaftliche begutachtungsprozesse. In *GI-VOI-BITKOM-OCG-TeleTrusT Konferenz "Elektronische Geschäftsprozesse" (eBusiness Processes), 24-25 September, Universität Klagenfurt (Austria)*, 2001. http://syssec.uni-klu.ac.at/EBP2001/.
5. B. Steffen and T. Margaria. Coarse-grain component based software development: The metaframe approach. In *3. Fachkongress "Smalltalk und Java in Industrie und Ausbildung" (STJA'97), Erfurt (Germany)*, September 10-11 1997.
6. B. Steffen and T. Margaria. Metaframe in practice: Intelligent network service design. In B. Steffen E.-R. Olderog, editor, *Correct System Design – Issues, Methods and Perspectives*, volume 1710 of *Lecture Notes in Computer Science (LNCS)*, pages 390 – 415, Heidelberg, Germany, 1999. Springer-Verlag.
7. Andreas Zeller. Making students read and review code. In *Proc. 5th ACM SIGCSE/SIGCUE Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '2000), Helsinki, Finland*, pages 89–92, 2000.

Demonstration der Problematiken beim Einsatz von Komponenten anhand eines Lernkurses aus JavaBeans-Komponenten

Ursula Scheben und Arnd Poetzsch-Heffter

FernUniversität Hagen, Feithstraße 142, D-58084 Hagen

1 Einführung und Motivation

Die Erwartungen an eine hohe Wiederverwendbarkeit von Software beim Einsatz objektorientierter Sprachen hat sich nicht erfüllt. Obwohl diese Sprachen gute Mechanismen für Datenabstraktion und Datenkapselung bereitstellen, haben sie nicht zur vermehrten Entwicklung unabhängiger, wiederverwendbarer Einheiten geführt. Statt dessen sind häufig große, monolithische Programme entstanden.

Es hat sich gezeigt, dass Klassen zu fein granular sind, um weitgehend unabhängige, wiederverwendbare Komponenten mit komplexerer Funktionalität zu bilden. Zur Bereitstellung solcher Funktionalitäten wird eine Menge von kooperierenden Klassen benötigt. Gute Konzepte zur Bildung größerer Einheiten fehlen aber in den meisten objektorientierten Sprachen.

Außerdem führt Vererbung -der Mechanismus zur Wiederverwendung in objektorientierten Sprachen- zum sogenannten *fragile base class*-Problem [12, 17]. Änderungen an einer Superklasse können unvorhersehbare Folgen für die Subklassen nach sich ziehen (Anpassungen, Neukompilierung, Sematikänderungen...).

Diese Probleme führten zur Entwicklung von Konzepten für besser wiederverwendbare Softwarebausteine, sogenannte *Komponenten* [9,13,17]. Komponenten sind (binäre) Einheiten bei denen Entwicklung, Erwerb und Einsatz unabhängig voneinander erfolgen und die interagieren, um ein funktionierendes System zu bilden. Komponenten kommunizieren über wohldefinierte Schnittstellen. Ihre Implementierung bleibt verborgen. Sie können sowohl objektorientiert mit Hilfe mehrerer Klassen implementiert werden als auch prozedural (siehe z.B. DCOM [7, 9, 17]).

Der Einsatz von Komponenten verspricht folgende Vorteile:

- Bessere Erweiterbarkeit von Applikationen durch Hinzufügen neuer Komponenten
- Bessere Anpassbarkeit von Applikationen an geänderte Bedürfnisse durch Austauschbarkeit von Komponenten (Beispiel: Komponente zur Rechtschreibprüfung für verschiedene Sprachen, von verschiedenen Anbietern)
- Höherer Grad an Wiederverwendbarkeit für andere Applikationen
- Möglichkeit zum Zukauf von Komponenten
- Reduzierung der Entwicklungskosten durch Zukauf oder Wiederverwendung
- Verteilung von Komponenten auf verschiedene Rechner
- Programmiersprachen- und Plattformunabhängigkeit
- Hohes Abstraktionsniveau (Bereitstellen von Geschäftskomponenten)
- Reduzierung der Komplexität bei der Erstellung einer neuen Applikation aus bestehenden Komponenten (Unterstützung nicht spezialisierter Entwickler)
- Niedrige externe Kopplung zwischen verschiedenen Komponenten

Wesentlich für die Akzeptanz von Komponenten ist, dass der Kompositionsvorgang (das ist das Zusammensetzen von Komponenten zu einem Gesamtsystem -auch *Assembly* genannt-) einfach ist und auch von Nicht-Programmierern durchgeführt werden kann. Dazu gehören u.a. eine grafische Unterstützung durch geeignete Werkzeuge, eine weitgehend automatische Anpassung von Komponenten an die Bedürfnisse ihrer Klienten sowie die Überprüfung der Gesamtkonfiguration auf Konsistenz.

Um zu sehen, welche Schwierigkeiten beim Zusammensetzen von Komponenten mit gängigen Werkzeugen auftreten, wurden an der FernUniversität Hagen Multimedia-Komponenten zum Aufbau von Fernstudienkursen entwickelt.

2 Multimedia-Komponenten für Studienzwecke

Für die Implementierung der Multimedia-Komponenten wurde das JavaBeans - Komponentenmodell [9, 17] gewählt.

Die entwickelten Komponenten sind eingeteilt in atomare und Containerkomponenten. Containerkomponenten dienen im wesentlichen dazu, atomare und andere Containerkomponenten aufzunehmen, zu gruppieren und zu verwalten sowie ggf. Konsistenzprüfungen durchzuführen. Zu atomaren Komponenten können keine anderen Komponenten hinzugefügt werden. Sie dienen zum Laden, Speichern und Darstellen von Informationen wie Texten, Bildern, Grafiken etc. und zum Steuern von Aktionen wie z.B. dem Starten eines Applets oder dem Starten einer Entwicklungsumgebung zum Bearbeiten von Programmierbeispielen.

Ein Fernstudienkurs setzt sich aus mehreren Kurseinheiten zusammen. Jede Kurseinheit kann ein Inhaltsverzeichnis, mehrere Kapitel, ein Stichwortverzeichnis, Literaturverzeichnis sowie Übungen und zugehörige Lösungen enthalten. Jedes Kapitel wiederum kann aus anderen Kapiteln sowie Paragraphen und atomaren Komponenten wie Text-, Bild-, Applet-, Programmierbeispiel- und Rechenkomponenten bestehen.

Kurseinheiten, Kapitel und Paragraphen sind Containerkomponenten. Die atomaren Komponenten sind eingeteilt in aktive und inaktive Komponenten. Beispiele für aktive Komponenten sind Komponenten, die Applets steuern, zur Bearbeitung von Programmierbeispielen dienen oder Taschenrechnerfunktionalitäten implementieren. Inaktive Komponenten sind z.B. Komponenten zum Anzeigen von Texten, Bildern und Grafiken.

Das Zusammensetzen der Komponenten zu einem Kurs erfolgte mit Hilfe der Beanbox [11], ihre Entwicklung mit Hilfe des JBuilders [8, 18].

3 Probleme und Lösungsansätze

Bei der Entwicklung und dem Einsatz der Multimedia-Komponenten wurden Probleme erkannt, die grob in zwei Kategorien eingeteilt werden können:

1. Probleme zwischen Buildertool und Komponenten
2. Probleme bei der Verbindung von Komponenten untereinander

Probleme der ersten Kategorie ergeben sich im wesentlichen daraus, daß das Buildertool Eigenschaften der Komponenten auswertet, die über die durch das Komponentenmodell festgelegten Eigenschaften hinausgehen. Dadurch können manche Komponenten in einem bestimmten Tool nicht oder nur bedingt verwendet werden.

Probleme der zweiten Kategorie haben u.a. folgende Ursachen:

- Der Entwickler (Autor) muss wissen, welche Komponenten verwendet werden können, um Applikationen eines bestimmten Typs (z.B. einen Fernstudienkurs) zu erstellen und wie sie zu verbinden sind. Das Buildertool unterstützt ihn dabei nicht.
- Er muss wissen, welche Verbindungen nicht erlaubt sind. Zum Beispiel darf eine Kurseinheit nicht zu einem Kapitel hinzugefügt werden. Das Buildertool führt eine solche Fehlkonfiguration ohne Warnung durch.
- Dem Entwickler muss bekannt sein, ob eine Komponente andere Komponenten benötigt, um ihre Aufgabe erfüllen zu können und wie die Verbindung zwischen diesen Komponenten einzurichten ist. Wird eine benötigte Verbindung zwischen zwei Komponenten nicht oder falsch hergestellt, äußert sich dies später in einem Fehlverhalten. Das Buildertool gibt keinen Warnhinweis aus. Beispielsweise benötigt die entwickelte Rechenkomponente eine Prüfkomponenten, die das Ergebnis der Rechenkomponente überprüft und zurückmeldet, ob das übergebene Ergebnis korrekt war oder nicht. Damit die Komponente für Programmierbeispiele korrekt arbeitet, muss die angegebene IDE verfügbar sein und gestartet werden können u.s.w.
- Einige Buildertools (z.B. Beanbox) bieten keine Möglichkeit, neue (wiederverwendbare) Komponenten aus bestehenden Komponenten zusammenzusetzen. Andere Buildertools (z.B. JBuilder) verfügen zwar über diese Option, setzen hierfür aber die Fähigkeiten eines professionellen Entwicklers voraus. Gerade im Bereich der Autorensysteme wäre es aber wünschenswert, schnell und einfach immer wieder benötigte Einheiten wie z.B. ein Video zusammen mit Buttons zum Starten und Anhalten erstellen und später beliebig wieder verwenden zu können.
- Nicht speziell für einen bestimmten Anwendungsbereich entwickelte Komponenten können häufig nicht in anderen Bereichen eingesetzt werden. Hier fehlen automatische Adaptionsmechanismen, mit denen eine Komponente an ein geändertes Umfeld angepasst werden kann. Beispielsweise wäre es wünschenswert, eine Button-Komponente in einem Fernstudienkurs verwenden zu können, die nicht speziell dafür entworfen wurde.

Ziel ist es, ein Buildertool in die Lage zu versetzen, den Entwickler besser als bisher beim Kompositionsvorgang unterstützen zu können und dadurch auch ungeübten Benutzern zu ermöglichen, Applikationen aus Komponenten zusammenzusetzen. Insbesondere soll es ihn auf bestehende Fehler aufmerksam machen und automatische Adaptionen durchführen können. Dafür müssen die Komponenten zusätzlich zu den durch das Komponentenmodell vorgeschriebenen Informationen -wie z.B. die unterstützten Schnittstellen, Methodensignaturen etc.- weitere Informationen bereitstellen. Dazu gehören z.B. Angaben über benötigte Schnittstellen anderer Komponenten, ggf. wechselseitige Verbindungen zweier Komponenten, nicht erlaubte Verbindungen zwischen Komponenten, die verwendete Architektur usw.

Hier liegt das aktuelle Forschungsgebiet des Projekts EASYCOMP, an dem das Lehrgebiet "Praktische Informatik V" der FernUniversität Hagen beteiligt ist. Im Lehrgebiet werden Spezifikationstechniken gesucht, die die oben genannten zusätzlichen Informationen in geeigneter Weise beschreiben und die eine automatische Auswertung von Seiten eines Buildertools ermöglichen.

Literatur

1. Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig, and Manfred Broy. A Formal Model for Componentware. In Gary T. Leavens and Murali Sitaraman, editors,

Foundations of Component-Based Systems, pages 189–210. Cambridge University Press, New York, NY, 2000.
2. J. Bosch and S. Mitchell, editors. *Object-Oriented Technology: ECOOP'97 Workshop Reader*, volume 1375 of *Lecture Notes in Computer Science*. The University of York, Department of Computer Science, Springer Verlag, Juni 1997.
3. Jan Bosch. Adapting Object-Oriented Components. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, TUCS General Publications No 5, pages 13 – 21. Turku Centre for Computer Science, September 1997. ISBN: 952-12-0039-1 ISSN: 1239-1905.
4. Dan Brookshier. *Java Beans Developer's Reference*. New Riders Publishing, Indianapolis, 1997.
5. Martin Büchi and Emil Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, TUCS General Publications No 5, pages 23 – 32. Turku Centre for Computer Science, September 1997. ISBN: 952-12-0039-1 ISSN: 1239-1905.
6. Stefan Denninger and Ingo Peters. *Enterprise JavaBeans*. Addison-Wesley, München, 2000.
7. Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press Deutschland, Unterschleißheim, 1998. ISBN: 3-86063-459-3.
8. Neal Ford, Ed Weber, Talal Azzouka, Terry Dietzler, Jennifer Streeter, and Casey Williams. *Borland JBuilder 3 Unleashed*. SAMS Publishing, 1999. ISBN: 0-672-31548-3.
9. Volker Gruhn and Andreas Thiel. *Komponentenmodelle. DCOM, JavaBeans, Enterprise JavaBeans, CORBA*. Addison-Wesley, München, 2000.
10. K. De Hondt, C. Lucas, and P. Steyaert. Reuse Contracts as Component Interface Descriptions. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, TUCS General Publications No 5, pages 43 – 49. Turku Centre for Computer Science, September 1997. ISBN: 952-12-0039-1 ISSN: 1239-1905.
11. Cay S. Horstmann and Gary Cornell. *Core Java 1.1, Volume II - Advanced Features*. Sun Microsystems Press, Palo Alto, 1998. ISBN: 0-13-766965-8.
12. L. Mikhajlov and E. Sekerinski. The fragile Base Class Problem and Its Impact on Component Systems. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology: ECOOP'97 Workshop Reader*, volume 1375 of *Lecture Notes in Computer Science*, pages 353 – 363. Springer Verlag, Juni 1997. ISBN: 3-540-64039-8 ISSN: 0302-9743.
13. Oscar Nierstrasz and Markus Lumpe. Komponenten, Komponentenframeworks und Gluing. *HMD - Theorie und Praxis der Wirtschaftsinformatik*, 197:8 – 23, September 1997. ISBN: 3-89864-101-5 ISSN: 1436-3011.
14. Palle Nowack. Interacting Components - A Conceptual Architecture Model. In *ECOOP Workshops*, pages 66–67, 1999.
15. Joao Costa Seco and Luis Caires. A Basic Model of Typed Components. In *ECOOP*, pages 108–128, 2000.
16. Joao Pedro Sousa and David Garlan. Formal Modeling of the Enterprise JavaBeans Component Integration Framework. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 Formal Methods, World Congress on Formal Methods in the Development of Software Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 1281 – 1300. Springer Verlag, 1999.
17. Clemens Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, New York, 1998.
18. Peter Tabatt and Henry Wolf. *Java programmieren mit JBuilder4*. Software & Support Verlag GmbH, Frankfurt, 2001. ISBN: 3-935042-04-3.

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult `http://aib.informatik.rwth-aachen.de/` **or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,**
Email: `biblio@informatik.rwth-aachen.de`

95-11 *	M. Staudt / K. von Thadden: Subsumption Checking in Knowledge Bases
95-12 *	G.V. Zemanek / H.W. Nissen / H. Hubert / M. Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology
95-13 *	M. Staudt / M. Jarke: Incremental Maintenance of Externally Materialized Views
95-14 *	P. Peters / P. Szczurko / M. Jeusfeld: Business Process Oriented Information Management: Conceptual Models at Work
95-15 *	S. Rams / M. Jarke: Proceedings of the Fifth Annual Workshop on Information Technologies & Systems
95-16 *	W. Hans / St. Winkler / F. Sáenz: Distributed Execution in Functional Logic Programming
96-1 *	Jahresbericht 1995
96-2	M. Hanus / Chr. Prehofer: Higher-Order Narrowing with Definitional Trees
96-3 *	W. Scheufele / G. Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
96-4	K. Pohl: PRO-ART: Enabling Requirements Pre-Traceability
96-5	K. Pohl: Requirements Engineering: An Overview
96-6 *	M. Jarke / W. Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
96-7	O. Chitil: The ς-Semantics: A Comprehensive Semantics for Functional Programs
96-8 *	S. Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
96-9	M. Hanus (Ed.): Proceedings of the Poster Session of ALP'96 — Fifth International Conference on Algebraic and Logic Programming
96-10	R. Conradi / B. Westfechtel: Version Models for Software Configuration Management
96-11 *	C. Weise / D. Lenzkes: A Fast Decision Algorithm for Timed Refinement
96-12 *	R. Dömges / K. Pohl / M. Jarke / B. Lohmann / W. Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
96-13 *	K. Pohl / R. Klamma / K. Weidenhaupt / R. Dömges / P. Haumer / M. Jarke: A Framework for Process-Integrated Tools
96-14 *	R. Gallersdörfer / K. Klabunde / A. Stolz / M. Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996
96-15 *	H. Schimpe / M. Staudt: VAREX: An Environment for Validating and Refining Rule Bases

96-16 * M. Jarke / M. Gebhardt, S. Jacobs, H. Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

96-17 M. Jeusfeld / T. X. Bui: Decision Support Components on the Internet

96-18 M. Jeusfeld / M. Papazoglou: Information Brokering: Design, Search and Transformation

96-19 * P. Peters / M. Jarke: Simulating the impact of information flows in networked organizations

96-20 M. Jarke / P. Peters / M. Jeusfeld: Model-driven planning and design of cooperative information systems

96-21 * G. de Michelis / E. Dubois / M. Jarke / F. Matthes / J. Mylopoulos / K. Pohl / J. Schmidt / C. Woo / E. Yu: Cooperative information systems: a manifesto

96-22 * S. Jacobs / M. Gebhardt, S. Kethers, W. Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

96-23 * M. Gebhardt / S. Jacobs: Conflict Management in Design

97-01 Jahresbericht 1996

97-02 J. Faassen: Using full parallel Boltzmann Machines for Optimization

97-03 A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

97-04 M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

97-05 * S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

97-06 M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

97-07 P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

97-08 D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting

97-09 C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets

97-10 M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

97-13 M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

97-14 R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

97-15 G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

98-01 * Jahresbericht 1997

98-02 S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes

98-03 S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

98-04 * O. Kubitz: Mobile Robots in Dynamic Environments

98-05 M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems

98-07 M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

98-08 * H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen
98-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien
98-10 * M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases
98-11 * A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML
98-12 * W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web
98-13 K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation
99-01 * Jahresbericht 1998
99-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
99-03 * R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager
99-04 M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing
99-07 Th. Wilke: CTL+ is exponentially more succinct than CTL
99-08 O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 * Jahresbericht 1999
2000-02 Jens Vöge / Marcin Jurdziński: A Discrete Strategy Improvement Algorithm for Solving Parity Games
2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05 * Mareike Schoop: Cooperative Document Management
2000-06 * Mareike Schoop, Christoph Quix (Ed.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 * Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08 Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 * Jahresbericht 2000
2001-02 Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03 Thierry Cachat: The power of one-letter rational languages
2001-04 Benedikt Bollig / Martin Leucker / Michael Weber: Local Parallel Model Checking for the Alternation free μ-calculus
2001-05 Benedikt Bollig / Martin Leucker / Thomas Noll: Regular MSC languages
2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07 Martin Grohe / Stefan Wöhrle: An Existential Locality Theorem
2001-08 Mareike Schoop / James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09	Thomas Arts / Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.