Datalogisk Institut
KØBENHAVNS UNIVERSITET

Proceedings of the 3rd DIKU-IST Joint Workshop on Foundations of Software

Robert Glück and Masami Hagiya (Eds.)

Technical Report no. 07/07
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

Preface

These proceedings contain the contributions presented at the *3rd DIKU-IST Joint Workshop on Foundations of Software* held at Roskilde, Denmark, October 5-6, 2007. The workshop featured talks and discussions on domain-specific languages, logic and model checking, program complexity and optimization, reversible and bidirectional computing, and demonstrations of software prototypes.

After the success of the first two joint workshops, which took place at Dragør, Copenhagen (2005) and Shonan Village, Japan (2006), the 3rd DIKU-IST workshop took again place in Denmark. It aimed to provide a forum for presenting the latest research and promoting the research collaboration between the *Department of Computer Science (DIKU), University of Copenhagen*, and the *Graduate School of Information Science and Technology (IST), University of Tokyo*. In 2004, IST and the Faculty of Science at the University of Copenhagen entered a *Five-Year Academic and Student Exchange Agreement* during the State Visit of Queen Margrethe II to Japan. Today, both universities are partners in the *International Alliance of Research Universities* (IARU).

Computer science provides one of the keys to the technologies of the 21st century. Its applications have found their way into all areas of daily life, often unnoticed by their users, and software has become a decisive factor for commercial success in many areas of modern business and society. This series of workshops between DIKU and IST is devoted to the scientific foundations of software. Theory and practice of programming languages are very important and visible research fields at both research institutions in Copenhagen and Tokyo. The objective of these workshops is to give researchers and graduate students at both institutions the opportunity to exchange the latest research ideas and to jointly engage in outstanding international research.

The workshop had about 45 participants from Japan and Denmark. The organizers would like to thank all speakers, participants, and the local organizers for making this meeting both a successful and enjoyable event. Special thanks to John Andersen, Director of International Affairs at the University of Copenhagen, His Excellency Masaki Okada, Ambassador of Japan in Denmark, as well as Masato Takeichi, Professor at IST and Member of the Science Council Japan, for opening the 3rd DIKU-IST workshop and giving it this special attention.

Copenhagen and Tokyo, November 2007

Robert Glück
Masami Hagiya

Organization

The *DIKU-IST Joint Workshop on Foundations of Software* was organized by the Department of Computer Science, University of Copenhagen, together with the Graduate School of Information Science and Technology, University of Tokyo.

Meeting

Program committee:	Jørgen Bansler Andrzej Filinski Robert Glück (Co-Chair) Masami Hagiya (Co-Chair) Fritz Henglein Zhenjiang Hu Neil Jones Julia Lawall Masato Takeichi
Local arrangements:	Holger Bock Axelsen Poul Clementsen Marianne Henriksen Kenji Moriyama Michael Kirkedal Thomsen
Proceedings:	Morten Fjord-Larsen
Website:	Jesper Andersen

Sponsoring Institutions

This workshop was sponsored by The Danish Natural Science Research Council (FNU) and The International Alliance of Research Universities (IARU) fund of the University of Copenhagen. We gratefully acknowledge the local support of the Department of Computer Science, University of Copenhagen.

Table of Contents

Applications

Semantics and Complexity

Debugging and Termination

Optimization

Roskilde, Denmark (October 2007)

Opening Address

John E. Andersen

Director of International Affairs
University of Copenhagen

Dear Ambassador and participants of the *Third Joint Workshop on Foundations of Software*, organised by DIKU, the Department of Computer Science, University of Copenhagen, and the Graduate School of Information Science and Technology at the University of Tokyo. On behalf of the Rector of the University of Copenhagen, I hereby warmly welcome you to the third – and hopefully not the last – Joint Workshop on Foundations of Software.

Our wish to enhance cooperation between the University of Copenhagen and the University of Tokyo has been at the top of our agenda for some time here in Copenhagen – and it goes without saying that the Rector welcomes this initiative to bring researchers together to exchange ideas and promote joint research in your field.

As a linguist and a Nordic philologist, I must say that I cannot fully understand the topics to be discussed here in the next couple of days – but some of the presentations do have very catchy titles, at least for me: for example, "*A Language for Dice Rolls*", "*Hunting Bugs with Cocinelle*" and last but not least "*Hiding Backtracking Operations*". I would imagine that in every field it could be useful to hide your backtracking operations.

In terms of enhancing collaboration between the University of Copenhagen and the University of Tokyo, there has been absolutely no backtracking. I actually just came back last week from a conference at the University of Tokyo, Hongo Campus, also attended by our Pro-rector Lykke Friis. During our stay, we met with your president Professor Komiyami – and when he learned that I would have the privilege of addressing you today, he asked me to convey his best regards, and he wishes you fruitful and pleasant encounters during the workshop.

You might ask what took us to Japan and the University of Tokyo – and the answer is IARU – *The International Alliance of Research Universities* – comprising 10 major universities worldwide.

As the Director of International Affairs, I am also the contact person for IARU at the University of Copenhagen. Since its inauguration early in 2006, cooperation between the ten partners has been very intense compared to most other alliances – we work together on a number of research projects. The project led by Tokyo is on "*Energy, Resources and Environment*". Sustainable cities with Tokyo as a model. By the way, just having come from Tokyo, I fully realise what a difference our colleagues from the University of Tokyo must experience being here in the middle of the Danish province – on top of the jet lag.

The IARU has also created joint programmes in the educational field – for example, summer schools led by Yale University and the National University of Singapore, a programme called *Global Summer Programme* (GSP), to be fully developed starting in 2009, where students and young researchers can come together for intensive educational programmes. I understand that you are looking for ways to continue your

joint workshops; one possibility might be to hoist the IARU sail and seek funding from that programme.

It is well known that Japanese universities are extremely hard to get into – and the University of Tokyo might be the hardest – but it is also said that once you are in, you do not have to do anything more. However, this was not what we witnessed. At your campus, the Japanese students seemed to have three hobbies: studying, studying and studying. And the Tokyo students not only know how to earn the top grades and get into the most prestigious university, they are also very concerned about climate change, sustainability and carbon footprints, they know martial arts – at least they can do a couple of deterrent kicks – they can use chopsticks to eat their soup, and their hairstyles are always low maintenance. Quite hard to compete with!

Although we are not at a University of Copenhagen campus, please allow me to say a few words about our University.

Our classical European university was founded in 1479 by permission of the Pope in Rome to the Danish King. At that time, there were sixty students and six professors – today there are about 38,000 students, 7,500 employees and eight faculties – the last two of which came on board this year by a merger with our agricultural university and our pharmaceutical university. If you have a chance to visit our main building, you will see high above the main gate a sculpture of an eagle. Below the eagle – written in the university lingua franca of the 19th century – are the words "*Coelestem Adspicit Lucem*", meaning "it sees the heavenly light". And this can be translated as: we are not in it for the money – here, we stick to the ivory tower and do blue sky research. Or at least those were the core values of a major university in the last century.

In the 21st century, this is not how we see ourselves. We value partnerships between public institutions and private companies and industries. We make the eagle occasionally look at the bottom-line without forgetting freedom – the truth-seeking research of the blue skies.

Thank you once more to the organisers of this conference, who have created the opportunity for many more interactions and increased cooperation between our two universities. At my office we have had, in particular, very fruitful cooperation with Associate Professor Robert Glück, who relentlessly pursues international action and contacts for DIKU. And may I take this opportunity to thank His Excellency, the Japanese Ambassador to Denmark, Mr. Masaki Okada, whom we consider a good friend of the University of Copenhagen, for showing such a keen interest in promoting cooperation between Denmark and Japan in the field of higher education and research.

The world has become flat – at least this is what they say about the global world and the global market – I hope that you will have a chance to see for yourself that Denmark has been flat for many thousands of years. You will probably not find the fish market in Roskilde as colourful, big or busy as the famous Tsukiji fish market in Tokyo, but the ships here – the Viking ships – are second to none. Take a look, and try to imagine how it would have been to cross the oceans on board one of these giant warships.

Enjoy your stay here – and enjoy the workshop!

Opening Address

Ambassador Masaki Okada

Embassy of Japan in Denmark

I am very happy to be able to witness the cooperation activities that are going on between Japanese and Danish researchers in the most advanced area of computer science. Both Japan and Denmark need to enhance their scientific capabilities and strengthen their status in high-tech fields in order to cope with the severe competition in the globalized world. Seen from Japan's side, in the past we have tended to concentrate our efforts on contact with American researchers, and this has probably meant that we have not paid due attention to the possibilities of cooperation with European scientists.

It was therefore very timely that, during his visit to Japan in November last year, the Danish Prime Minister, Mr. Rasmussen, proposed strengthening the exchange between universities and research institutions of the two countries. The exchange between the University of Tokyo and the University of Copenhagen in the area of computer science is a very good example of the new cooperation, and I hope that this case will become a precedent for numerous cooperation projects between Japan and Denmark in the years to come.

I sincerely hope that the workshop will deepen the friendship between Japanese and Danish scientists and expand cooperation in the field of science and technology in the future.

DIKU-IST Collaboration
and
Joint Workshops on
Foundations of Software

Professor Masato Takeichi
School of Information Science and Technology(IST)
University of Tokyo

Academic Exchange Agreement Ceremony
between Faculty of Science, University of Copenhagen
and Graduate School of IST, University of Tokyo

Her Majesty the Queen meets the researchers in Tokyo

Works after the 1st DIKU-IST Workshop at Copenhagen

September 24, 2005

Participants of the 2nd DIKU-IST Workshop at Shonan-Village

April 21-22, 2006

View from Comwell Conference Site of the 3rd DIKU-IST Workshop at Roskilde, Denmark

October 5-6, 2007

President Komiyama of the University of Tokyo and Rector Hemmingsen of the University of Copenhagen at IARU Presidents' meeting, Singapore, January 2006

International Alliance of Research Universities
The Australian National University, ETH Zurich, National University of Singapore , Peking University, University of California, Berkeley, University of Cambridge, University of Copenhagen, University of Oxford, The University of Tokyo, Yale University

Opening - 3rd DIKU-IST Joint Workshop on Foundations of Software

Fri & Sat, 5-6 October 2007, Comwell Hotel, Roskilde, Denmark

John Andersen
(Director of International Affairs, University of Copenhagen)

Masaki Okada
(Ambassador of Japan In Denmark)

40+ participants
(11 from IST, University of Tokyo)

A Language for Optimal Path Queries

Akimasa Morihata
(IST, University of Tokyo)

3rd DIKU-IST Joint Workshop on Foundations of Software,
October 2007, Roskilde, Denmark

1

Can you solve this problem?

You are planning a trip.
You want to visit some temples and museums.
You may go by walking, train, bus, etc...
Your budget is fixed.
Traffic jam may trouble you in afternoon.
Climbing a temple will make you tired.
Find the shortest route of the tour.

2

Optimal paths are asked everywhere!

Optimal path query:
Finding one of the minimum-weighted paths subject to some constraints

Optimal path queries are important

- to plan trips
- to arrange schedules
- to design networks
- to extract data from databases [Flesca et al. 06]

3

From theories to a language

Status: rich theoretical results

- resource constrained SP [Handler & Zang 80]
- regular-language constrained SP [Romeuf 88]
- SP in time-dependent networks [Orda & Rom 90]

Non-specialists hardly enjoy these results

Proposal:
a *language* for optimal path queries

4

Graphs and paths

- Directed graph $G = (V, E)$
  - Weight functions $w_i : E \to \mathbb{N}$
  - Label functions $l_i : E \to \Sigma$
- Path = sequence of edges
  - $xa \stackrel{\text{def}}{=} x +\!\!+ [a]$

5

Definition of our language

$$
\begin{array}{rcll}
prog & ::= & \text{minimize } f_0(x) \text{ subject to } p_0(x) & \\
 & & \text{where } decl \cdots decl & \\
decl & ::= & p_k(\epsilon) = b;\ p_k(xa) = \phi; & \\
 & | & f_k(\epsilon) = n;\ f_k(xa) = f_k(x) + e; & \\
\phi & ::= & b \mid \neg\phi \mid \phi \wedge \phi \mid l_i(a) = \sigma \mid p_i(x) & \\
 & | & e \leq n & n \in \mathbb{N}, \\
e & ::= & n \mid e + e \mid w_i(a) \mid f_i(x) & b \in \{\mathsf{T}, \mathsf{F}\}, \\
 & | & \text{if } \phi \text{ then } e \text{ else } e & \sigma \in \Sigma
\end{array}
$$

f_0 must not call other functions outside logical expressions

6

Example 1:
Shortest path from s to t via c

minimize $f_0(x)$ subject to $p_0(x)$ where

$$
\begin{aligned}
f_0(\epsilon) &= 0; \\
f_0(xa) &= f_0(x) + w(a); \\
p_0(\epsilon) &= \mathsf{F}; \\
p_0(xa) &= startS(x) \wedge hasC(x) \wedge out(a) = \mathtt{t}; \\
hasC(\epsilon) &= \mathsf{F}; \\
hasC(xa) &= hasC(x) \vee in(a) = \mathtt{c} \vee out(a) = \mathtt{c}; \\
StartS(\epsilon) &= \mathsf{F}; \\
StartS(xa) &= StartS(x) \vee (empty(x) \wedge in(a) = \mathtt{s}); \\
empty(\epsilon) &= \mathsf{T}; \\
empty(xa) &= \mathsf{F};
\end{aligned}
$$

Example 2:
We will get tired after walking a lot

minimize $f_0(x)$ subject to $p_0(x)$ where

$$
\begin{aligned}
f_0(\epsilon) &= 0; \\
f_0(xa) &= \sout{f_0(x) + w(a);} \\
p_0(\epsilon) &= \mathsf{F} \\
p_0(x &\\
hasC &\\
hasC(xa) &= hasC(x) \vee in(a) = \mathtt{c} \vee out(a) = \mathtt{c}; \\
StartS(\epsilon) &= \mathsf{F}; \\
StartS(xa) &= StartS(x) \vee (empty(x) \wedge in(a) = \mathtt{s}); \\
empty(\epsilon) &= \mathsf{T}; \\
empty(xa) &= \mathsf{F};
\end{aligned}
$$

$f_0(x) + (\text{if } f_0(x) \leq 15 \text{ then } w(a) \text{ else } w(a) + w(a));$

Properties of the language

Lemma: $f_i(x) \leq f_i(xa)$ for all f_i

We want to obtain a function s such that

$$
s(x) = s(y) \Rightarrow \begin{cases} s(xa) = s(xa) \\ p_0(x) = p_0(y) \\ f_0(x) \leq f_0(y) \Rightarrow f_0(xa) \leq f_0(ya) \end{cases}
$$

$s(x) \stackrel{\text{def}}{=} (p_1(x), \ldots, p_n(x), f'_0(x), \ldots, f'_m(x), \underline{d(x)})$

destination

- $f'_i(x) \stackrel{\text{def}}{=} \min(f_i(x), u_{f_i})$
- $u_{f_0}, \ldots, u_{f_m}$: max values distinguished by "$\leqq$"

9

Branch-and-bound algorithm for optimal path query

$N = \emptyset,\ W = E;$
while $(W \neq \emptyset)$ {
  $x = min_{f_0}(W), N = N \cup \{s(x)\}, W = W \setminus \{x\};$
  if $p_0(x)$ return x;
  foreach xa s.t. $s(xa) \notin N$ {
    if $(\forall y \in W : s(xa) \neq s(y))$
      $W = W \cup \{xa\};$
    if $(\exists y \in W : s(xa) = s(y) \wedge f_0(xa) < f_0(y))$
      $W = W \setminus \{y\} \cup \{xa\};$
  }
}
return $\perp$;

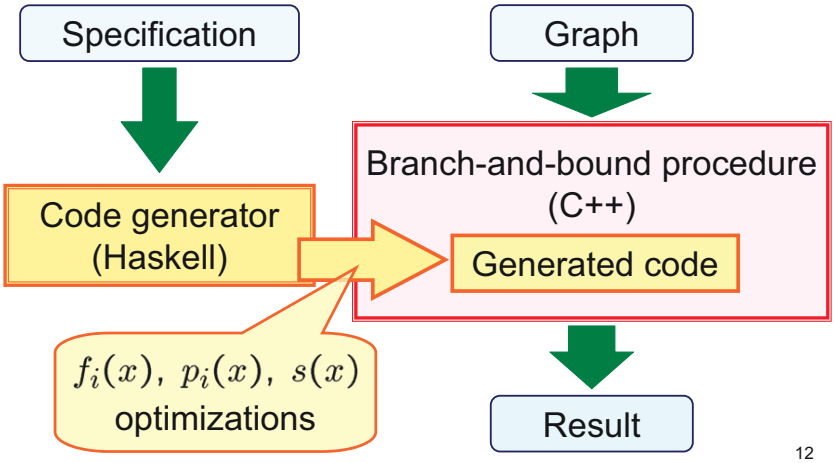
Note: this algorithm always terminates

Optimizations: memoizaion and unnecessary-candidate removal

- Memoization
  – memoizing each result of functions/predicates
- Unnecessary-candidate removal
  e.g. find a path from v to t
  → paths not starting from v are unnecessary
  – constructing a predicate to check necessity
    - Reachablity checking on the transition graph of $s(x)$

11

Implementation



12

Related works

- Variants of shortest path problems (quite many)
- Route planner of TRANSIMS project [Barrett et al., 2000—2007]
  - regular-language constrained, time-dependent shortest path queries
- Derivation of dynamic programming algorithm [Sasano et al., 2000—2006]

13

Conclusion and future works

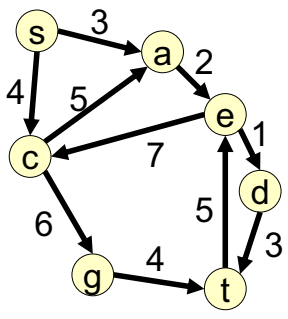
A language for optimal path queries
- expressive
- an efficient algorithm with optimizations

- Implementation and experiment
  - expressive/efficient enough for practical uses?
- Good external language
- Further optimizations

14

Small demo

1. Shortest path from s to t via c
2. ...etc? (if time is left)



15

Time Complexity

V = (number of verties)

E = (number of edges)

kV = (size of the range of s)

k is exponential to the size of the specification

$O(kE \log(kV))$:

current implementation (by binary heap)

$O(kV \log(kV) + kE)$:

theoretical complexity (by Fibonacci heap)

16

Troll: A Language for Dice-Rolls

Torben Mogensen, DIKU

DIKU-IST 2007

Why have a DSL for dice-rolls?

- Concise and unambiguous descriptions for communicating betwen people.
- Internet dice servers.
- Probability calculations for
  - Figuring your chances (player).
  - Deciding difficulty level (GM).
  - Design space exploration (game designer).

Notation for dice-rolls – from D&D to **Troll**

- The role-playing game "Dungeons & Dragons" from 1974 introduced use of non-cubical dice

- . . . and notation such as `3d10+2`. This notation has been used in many later games.
- Many games use dice-rolls that can't be described by the notation from D&D.
- In 2002 I designed **Roll** as an attempt at a universal notation for dice-rolls and made programs for rolling and analysing rolls described in **Roll**.
- **Roll** was used in the design of the latest version of the game "World of Darkness" from 2004.

- Some dice-rolls were not easy to describe in **Roll**, so in 2006 I made the successor **Troll**.

Elements of **Troll**

- A roll is a *collection* (multiset) of numbers:
  - Order is irrelevant
  - Number of occurences is significant.
- A collection with one element can be used as a number.
  Some operations require this.
- Collections can be combined, filtered, counted, summed and in other ways manipulated to find a final result.
- Two different semantics:
  - Random rolling
  - Calculation of probability distribution

Basic **Troll** operations

- `d`*N* rolls a single *N*-sided die.
- *M*`d`*N* rolls *M* *N*-sided dice and makes a collection of the results.
- `sum` *C* adds the elements in the collection *C*.
- `counts` *C* counts the elements in the collection *C*.
- `+`, `-`, `*`, `/` do arithmetic on numbers.
- `@` finds the union of two collections.
- *M* `<` *C* returns the elements of *C* that are greater than *M*. Also for `=`, `>`, `<=`, `>=`, `=/=`.
- `min` and `max` find the smallest or largest element in a collection, respectively.
- `least` *N* and `largest` *N* find the least or largest *N* elements of a collection.

Simple `Troll` definitions

- `sum 2d10 + 3`
  Adds two ten-sided dice and adds 3 to the result.
- `sum largest 3 4d6`
  adds the largest 3 of 4 six-sided dice.
- `count >7 6d10`
  counts how many out of six d10s are greater than 7.
- `max 3d20`
  finds the largest of three d20.

Advanced features

- M `#` e makes M *independent* samples of expression e and combines the results using `@`.
- `if` C `then` e_1 `else` e_2 If C is non-empty, do e_2, otherwise do e_3.
- x `:=` e_1`;` e_2 defines x to be the value of e_1 inside e_2. Note: Only one sample of e_1.
- `repeat` x `:=` e_1 `until` e_2 repeats rolling e_1 until the expression e_2 evaluates to non-empty, then returns last value of e_1.
- `accumulate` x `:=` e_1 `until` e_2 repeats rolling e_1 until the expression e_2 evaluates to non-empty, then returns the union of all values of e_1.
- `foreach` x `in` e_1 `do` e_2 calculates e_1, and for each number in the result evaluates s e_2 with x bound to that number, then unions the results of e_2.

Advanced examples

- `b := 2d6; if min b = max b then b@b else b`
  Backgammon dice.
- `count 7< N#(accumulate x:=d10 while x=10)`
  Die roll for *World of Darkness*.
- `repeat x := 2d6 until (min x) < (max x)`
  Roll two d6 until you don't have a double.
- `x := 7d10; max foreach i in 1..10 do sum i= x`
  Largest sum of identical dice.

Implementation

- Implemented in Moscow ML.
- The two semantics:

  Random rolls is implemented fairly straightforwardly using a PRNG.

  Probability distribution implemented by enumerating all possible rolls (with optimisations) and counting results.

Optimizations

Exploits *linear* and *homomorphic* functions:

Linear: $f(C_1 @ C_2) = f(C_1) @ f(C_2)$
Examples: `M<`, `M=`, `foreach`

Homomorphic: Exists $\oplus$ so $f(C_1 @ C_2) = f(C_1) \oplus f(C_2)$
Examples: `sum`, `count`, `min`, `least N`, `if`, `different`

Uses unnormalized representation of probability distributions:

```
datatype dist = VAL of value
              | CHOICE of real * dist * dist
              | UNION of dist * dist
              | TWICE of dist
```

Exploiting unnormalized distribution

Linear `f`:

```
fun linear f (VAL v)            = VAl (f v)
  | linear f (CHOICE (p,d1,d2)) = CHOICE (p, linear f d1, linear f d2)
  | linear f (UNION (d1,d2))    = UNION (linear f d1, linear f d2)
  | linear f (TWICE d)          = TWICE (linear f d)
```

Homomorphic `h`:

```
fun hm h g (VAL v)              = VAL(h v)
  | hm h g (CHOICE (p,d1,d2))   = CHOICE (p, hm h g d1, hm h g d2)
  | hm h g (UNION (d1,d2))      = up g (hm h g d1) (hm h g d2)
  | hm h g (TWICE d)            = up2 g (hm h g d)
and up g (VAL v) (VAL w)        = VAL(g v w)
  | up g (CHOICE (p,d1,d2)) d3 = CHOICE (p,up g d1 d3, up g d2 d3)
  | up g d1 (CHOICE (p,d2,d3)) = CHOICE (p,up g d1 d2, up g d1 d3)
and up2 g (VAL v)               = VAL (g v v)
  | up2 g (CHOICE (p,d1,d2))    = CHOICE (p*p,up2 g d1,
                                    CHOICE ((1-p)*(1-p)/(1-p*p),
                                          up2 g d2,
                                          up g d1 d2))
```

Other optimizations

Exploit that `repeat` and `accumulate` have unchanged conditions in all iterations:

- Distribution of body calculated once, then rewritten into the form

  $$\texttt{CHOICE}(p_{fail}, d_{fail}, d_{succ})$$

  where the values in d_{fail} fail the condition and values in d_{succ} succeed.
- For `repeat-until`, the resulting distribution is d_{succ}.
- For `accumulate-while`, the resulting distribution d is given by the equation

  $$d = \texttt{CHOICE}(p_{fail}, d_{fail}, \texttt{UNION}(d_{succ}, d))$$

  Solution is infinite, but cut off after specified limit.

Experiences with **Troll**

- Non-programmers can write simple definitions.
- While optimisations help a lot, sometimes **Troll** needs to enumerate all combinations, which may be slow.
- New features added occasionally by request from users (latest: text and recursive function definitions).
- Download from `www.diku.dk/~torbenm/Troll` (Requires Moscow ML).

Modal μ-calculus on min-plus algebra $\mathbf{N}_\infty$ and its applications

Masami Hagiya
Joint work with
Yoshinori Tanabe
Koki Nishizawa
Dai Ikarashi

Contents

- Background: modal μ-calculus, min-plus algebra, etc.
- Interpretation of formulas on min-plus algebra and examples
- Algorithm for model checking (current status report)
- Application to shape analysis
- Related work and future work

Modal μ-calculus: For what?

- Used to describe properties to be verified in model checking (a kind of modal/temporal logic)
- Model checking: regard software or hardware as a state transition system and verify its properties by exhaustive state exploration
- Examples of model checkers: SPIN, NuSMV, JavaPathFinder

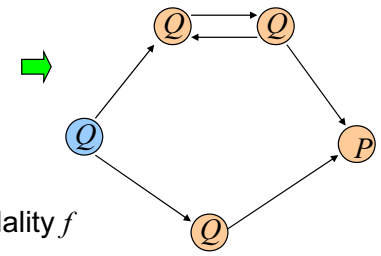
What is modal μ-calculus?

- Modal logic: logical system obtained by adding [ ] and ⟨ ⟩ to propositional logic
  - $[f]P$ --- Necessarily P w.r.t. modality f
  - $\langle f \rangle P$ --- Possibly P w.r.t. modality f
- Modal μ-calculus: logical system obtained by adding the least fixed point operator μ and the greatest fixed point operator ν

Kripke Structure

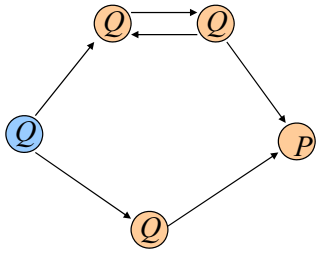
- Assume a fixed set of "atomic propositions"
- Kripke structure: state transition system where each state has atomic propositions that are true at the state

Example of a Kripke structure
P and Q are atomic propositions
Only true formulas are shown
The initial state is in light blue
Arrows denote transitions by modality f



Examples of Formulas

- Formula $\mu X(P \vee \langle f \rangle X)$
  - A state where P is true is reachable via transitions by f
- Formula $\nu X(P \vee (Q \wedge [f]X))$
  - In all transition sequences by f, Q always holds while P is not



What is min-plus algebra?

- Also known as min-plus dioid
- Used to analyze and optimize discrete event systems

Definition of a dioid

- **min** is associative and commutative, has zero element ∞, and is idempotent
- Zero element is absorptive w.r.t. **plus**
- **plus** is associative, communative, has unit element 0, and distributes over **min**

※**min** corresponds to addition, **plus** to multiplication of rings or fields

Why min-plus algebra?

- Not only truth values, but numerical measures become available by introducing **plus**
- Applications:
  - Verification of liveness in shape analysis
  - Flow analysis in compilers
  - …

Why min-plus? --- Shape Analysis

- Shape analysis: verification of algorithms operating on data structures in heap, such as lists, trees, DAGs, etc.
- Directed graphs can be regarded as Kripke structures if atomic propositions true on each state are defined
  - Shape analysis using modal logic (Tanabe et al. 2005)

Shape Analysis (cont.)
Size-change principle

- Size-change termination principle:
  - If some measure decreases in each iteration of a loop, it eventually terminates in a finite number of steps
- By introducing **plus**, numerical measures can be defined and computed
  - Measures before and after an operation on heap are compared and the size-change principle is applied (later in the talk)

Semantics

- **plus** → $\wedge$
- **min** → $\vee$
- For formula φ

  $[\varphi] : S \to \mathbf{N} \cup \{\infty\}$

  S : the set of all sates in a Kripke structure

  $\mathbf{N} = \{0, 1, 2, \ldots\}$

  $\mathbf{N} \cup \{\infty\}$ $(= \mathbf{N}_\infty)$: min-plus algebra

$[P]x = 0$ **if** $x \models P$ $(x \in S)$

$= \infty$ **otherwise**

P : atomic proposition

$[\varphi \vee \psi]x = \mathbf{min}([\varphi]x, [\psi]x)$

$[\varphi \wedge \psi]x = [\varphi]x + [\psi]x$

$[\neg\varphi]x = \infty$ **if** $[\varphi]x < \infty$

$= 0$ **if** $[\varphi]x = \infty$

$[\langle f \rangle \varphi]x = \mathbf{min}\{\ [\varphi]y \mid R_f(x, y)\ \}$

$[[f]\varphi]x = \sum\{\ [\varphi]y \mid R_f(x, y)\ \}$

$R_f(x, y)$: state y is reachable from x by modality f

$[1]x = 1$ **for any** $x \in S$

$[\mu X\, \varphi] = \mathbf{max}\{\ F : S \to \mathbf{N}_\infty \mid F = [\varphi]_{[X]:=F}\ \}$

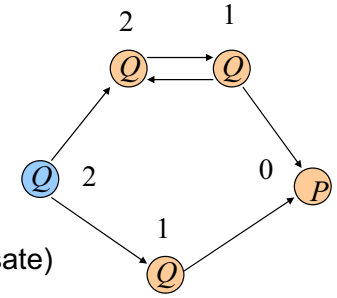
X : propositional variable

Example

- Formula $\mu X(P \vee \langle f \rangle(1 \wedge X))$
  - The length of the shortest path from a state to some state satisfying P

On contrary: in the ordinary modal μ-calculus

- Formula $\mu X(P \vee \langle f \rangle X)$
  - Whether there exists a path from a state to some state satisfying P



- Formula $[o](\neg Q \vee 1)$
  - The number of states satisfying Q

(o denotes the global modality:
Any state is reachable from any sate)

(Local) Model Checking

- In the ordinary semantics, check whether φ holds at x (true or false) for formula φ and state x in the Kripke structure
- In the min-plus algebra semantics, compute the value of $[\![\varphi]\!]x \in \mathbf{N}_\infty$
- According to the construction of φ
  - Easy except for fixed points

Computation of Fixed Points

- The traditional iterative computation of fixed points assumes a finite complete lattice (e.g., finite-state and ordinary truth values)
- Even if the domain is not finite, if it is well-founded (no infinite decreasing sequences), the traditional approach can be directly applied to greatest fixed points
  - Since min-plus algebra semantics is well-founded for a finite-state Kripke structure, the μ-operator (greatest fixed point) can be computed
- However, the ν-operator may produce infinite increasing sequences and the computation may not terminate in a finite number of steps
  - New algorithm required!

Computing the ν-operator

- We have developed an algorithm for computing the ν-operator (least fixed point) under the restriction that μ, ν, $\supset$, $\neg$ do not occur in the scope of the ν-operator
  - We are now extending the algorithm for the unrestricted case
  - **After the workshop, we have already shown the decidability of model checking including** $\supset$, $\neg$

Standard Form of Semantics

- Consider a formula φ without μ, ν, $\neg$, $\supset$
  - In this case, the semantics of φ, $[\varphi]x$, contains only **min** and **plus**
- Since **plus** distributes over **min**, which can be pulled out of **plus**, $[\varphi]x$ can be expressed in the form

$$\mathbf{min} \sum(...)$$

Idea on ν-operator Computation

- Consider $\nu X\, \varphi$
  - X is the only propositional variable in φ
  - Let $L = S \to \mathbf{N}_\infty$ and $S = \{s_1, \ldots, s_n\}$
  - X ranges over L
  - Formula φ is interpreted as a function: $L \to L$
  - Write φX for the interpretation of φ with X
- By the previous slide

$$(\varphi X)s_i = \mathbf{min}_j \sum_k (A_{ijk} X s_k + C_{ij})$$

- Let $m_i = \mathbf{min}\{C_{ij} \mid \forall k\,.\, A_{ijk} = 0\}$
- Proposition: If F is the least fixed point of φ and $Fs_j \neq 0$ for each s_j, then $\mathbf{min}_j\, m_j = m_i$ implies $Fs_i = m_i$
  - The least fixed point is obtained by iterative computation

$Fs_j \neq 0$??

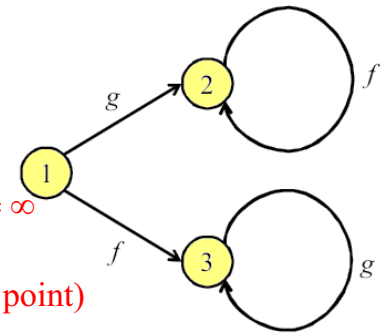
- In the previous slide, it was assumed that $Fs_j \neq 0$ for each s_j
- For this assumption to be true, those s_j s.t. $Fs_j = 0$ should be obtained in advance
  - Fixed point computation using abstraction
  - $0 \rightarrow 0$
  - $1, 2, \ldots, \infty \rightarrow \infty$
  - Ordinary fixed point computation determines whether Fs_j is 0 or not

Example

- Formula $\nu X\,(\langle f\rangle(1 \wedge X) \vee \langle g\rangle X)$
  - $\varphi X = [\langle f\rangle(1 \wedge X) \vee \langle g\rangle X]$
- Assume the Kripke structure on the right
  - $S = \{1, 2, 3\}$
- $\varphi X1 = \mathbf{min}(X2, X3 + 1)$
  $\varphi X2 = X2 + 1$
  $\varphi X3 = X3$

Step1: abstraction

- $\varphi' X1 = \mathbf{min}(X2, X3 + \infty) = \infty$
  $\varphi' X2 = X2 + \infty = \infty$
  $\varphi' X3 = X3 = 0$ (least fixed point)



Example

$F3 = 0$ is determined

Step2 starts with $F1 = \infty, F2 = \infty$

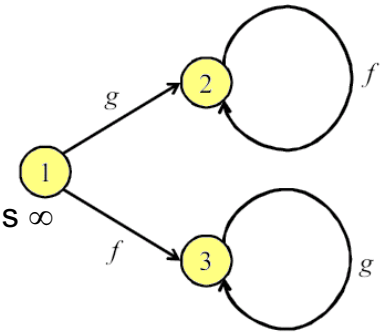
- Equations for $F1, F2$
  $\varphi F1 = \mathbf{min}(F2, F3 + 1) = 1$
  $\varphi F2 = F2 + 1 = \infty$

$F1 = 1$ is determined

- Equations for $F2$
  $\varphi F2 = F2 + 1 = \infty$

Stop since the least value is ∞

$F2 = \infty$ is determined



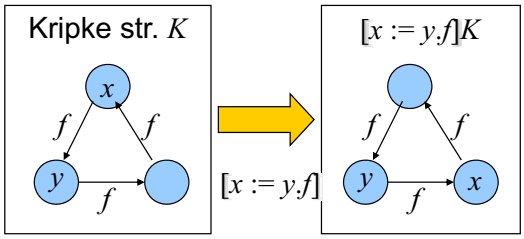
Functional Kripke Structure, Nominal

- Functional Kripke structure: Modalities (other than the global modality) relate exactly one target state for each state in transition
- Nominal: atomic proposition that is true at exactly one state
  - x, y denote nominals
    (each corresponding to a state)
  - $@x.\varphi$ abbreviates $@x.\varphi \equiv [o](x \supset \varphi) \equiv \langle o \rangle (x \wedge \varphi)$

Shape Analysis: Primitive Statements

- Five kinds of primitive statements

  $x := y$
  $x := y.f$
  $x.b := \text{true}$
  $x.b := \text{false}$
  $x.f := y$



- Primitive statement σ is interpreted as a function:

  $$[\sigma] : \mathcal{K} \to \mathcal{K}$$

  - $\mathcal{K}$ denotes the set of all functional Kripke structures

Weakest Precondition and Comparison of Formulas

- In order to apply the size-change principle, one has to show

  $$[\psi]^K s > [\psi]^{[\sigma]K} s$$

  - Different Kripke structures (K and $[\sigma]K$) on the left and right sides
- The weakest precondition of ψ for σ is a formula $\text{wp}(\sigma, \psi)$ satisfying

  $$[\psi]^{[\sigma]K} s = [\text{wp}(\sigma, \psi)]^K s$$

- Using wp, the above inequality becomes

  $$[\psi]^K s > [\text{wp}(\sigma, \psi)]^K s$$

  - The same Kripke structure on both sides
- $\text{wp}(\sigma, \psi)$ can be computed as in the next slide

表 1: $\mathrm{wp}(\sigma, \psi)$

ψ \ σ	x:=y	x:=y.f	x.b:=true	x.b:=false	x.f:=y
$\bot$	$\bot$				
b	b		$b \vee x$	$b \wedge \neg x$	b
b'	b'				
x	y	$\langle f^{-1} \rangle x$	x		
x'	x'				
i	i				
X	X				
$\varphi_1 \vee \varphi_2$	$\overline{\varphi_1} \vee \overline{\varphi_2}$				
$\varphi_1 \wedge \varphi_2$	$\overline{\varphi_1} \wedge \overline{\varphi_2}$				
$\varphi_1 \rightarrow \varphi_2$	$\overline{\varphi_1} \rightarrow \overline{\varphi_2}$				
$\langle f \rangle \varphi$	$\langle f \rangle \overline{\varphi}$				$(x \rightarrow @y.\overline{\varphi}) \wedge (\neg x \rightarrow \langle f \rangle \overline{\varphi})$
$\langle f^{-1} \rangle \varphi$	$\langle f^{-1} \rangle \overline{\varphi}$				$\langle f^{-1} \rangle (\neg x \wedge \overline{\varphi}) \vee (y \wedge @x.\overline{\varphi})$
$\langle f' \rangle \varphi$	$\langle f' \rangle \overline{\varphi}$				
$[f] \varphi$	$[f] \overline{\varphi}$				$(x \rightarrow @y.\overline{\varphi}) \wedge (\neg x \rightarrow [f] \overline{\varphi})$
$[f^{-1}] \varphi$	$[f^{-1}] \overline{\varphi}$				$[f^{-1}](x \vee \overline{\varphi}) \wedge (y \rightarrow @x.\overline{\varphi})$
$[f'] \varphi$	$[f'] \overline{\varphi}$				
$\mu X \varphi$	$\mu X \overline{\varphi}$				
$\nu X \varphi$	$\nu X \overline{\varphi}$				
$@x.\varphi$	$@y.\overline{\varphi}$	$@y.\langle f \rangle \overline{\varphi}$	$@x.\overline{\varphi}$		
$@x'.\varphi$	$@x'.\overline{\varphi}$				

Example

- $\varphi = @x.\mu X(a \vee \langle f \rangle (1 \wedge X))$, $\sigma = (x := x.f)$
  - We obtain $\mathrm{wp}(\sigma, \varphi) = @x.\langle f \rangle \mu X(a \vee \langle f \rangle (1 \wedge X))$
- Compare φ before and after σ
  - Assume that a does not hold at state x
  - Also assume $[\varphi]^K s < \infty$ for $s \in S$
  - We can then show $[\varphi]^K s > [\varphi]^{[\sigma]K} s$ by showing $[\varphi]^K s > [\mathrm{wp}(\sigma, \varphi)]^K s$ according to the semantics

Related Work

- Nishizawa et al. used ordinary complete Heyting algebra for truth values (Nishizawa et al. 2007)
  - Although min-max algebra is complete Heyting algebra, it cannot be directly compared with min-plus algebra
- Needs to be compared with more related work

Future Work

- Fixed point computation in general case
- Implementation of model checking algorithm
- Application of the ν-operator
- Application to shape analysis
- Application to flow analysis
- Extension to algebra with **max**

Future Work

- Fixed point computation in general case
- Implementation of model checking algorithm
- Application of the ν-operator
- Application to shape analysis
- Application to flow analysis
- Extension to algebra with **max**
  - In $\mathbf{N}_\infty$, $\mathbf{max}(\varphi, \psi) = ((\varphi \vee \psi) \supset \varphi) \wedge \psi$
  - $[\varphi \supset \psi]x =$ **the smallest** n **s.t.** $[\varphi]x + n \geq [\psi]x$

Practical Program Transformation using Temporal Logic and Model Checking

Julia Lawall

Joint work with René Rydhof Hansen, Jesper Andersen (DIKU), Yoann Padioleau, and Gilles Muller (EMN)

October 5, 2007

Device drivers: a notorious weak point in OS code

Developed by device-experts, not core kernel experts.

- Protocols and coding conventions not always understood.
- Drivers can fall behind the rest of the kernel.
- Errors in driver code lead to system crash.

Previous (and ongoing) work:

- Verification and bug detection [Ball, Engler, Foster, etc.]
- Protocol detection [Engler, Zhou, etc.]
- Safe or domain-specific languages [SPIN, Cyclone, Devil, etc.]

Our focus: automating device driver collateral evolution.

Collateral evolution

Collateral evolution: an update required in a library client in response to an evolution affecting the interface of a library.

In the case of Linux:

- Drivers rely heavily on internal Linux libraries.
- These libraries and their usage protocols are evolving rapidly.
- Updating the drivers that rely on these libraries (collateral evolution) is time-consuming and error prone.
- Updating drivers outside the Linux source tree additionally requires a transfer of expertise.

Examples

Yoann arrives at UIUC and wants to use their VPN:

- Drop `#include <config.h>`
- Drop the second argument of `skb_checksum_help`.
- Replace `CHECKSUM_HW` by either `CHECKSUM_PARTIAL` or `CHECKSUM_COMPLETE` (which? when?)

Examples

Yoann arrives at UIUC and wants to use their VPN:

- Drop `#include <config.h>`
- Drop the second argument of `skb_checksum_help`.
- Replace `CHECKSUM_HW` by either `CHECKSUM_PARTIAL` or `CHECKSUM_COMPLETE` (which? when?)

To access `Scsi_Host` information:

- Old protocol: call get, check the result, process, call put.
- New protocol: receive the information as an argument.
- Used by 19 scsi drivers in the Linux source tree.

Goals

Document and automate collateral evolutions.

Need a notation that:

- Is easy to write, easy to read. WYSIWYG approach.
- Expresses code-level transformations.
- Provides confidence in the result.
- Is acceptable to driver maintainers.
- Focuses on device-driver relevant issues (eg, simple structure, wide use of copy-paste).

Example: The Scsi_Host protocol in more detail

scsi_host_hn_get and scsi_host_put:

- Access and release a Scsi_Host typed structure.
- Manage a reference count. Dangerous.

Linux 2.5.71:

- scsi_host_hn_get and scsi_host_put no longer exported.
- Functions using them get Scsi_Host value as an argument.
- In practice, only affects proc_info functions.

A scsi driver: scsiglue.c (simplified proc_info function)

```
static int usb_storage_proc_info (
         char *buffer, char **start, off_t offset,
         int length, int hostno, int inout)
{
  struct us_data *us;
  struct Scsi_Host *hostptr;

  hostptr = scsi_host_hn_get(hostno);
  if (!hostptr) { return -ESRCH; }

  us = (struct us_data*)hostptr->hostdata[0];
  if (!us) {
    scsi_host_put(hostptr);
    return -ESRCH;
  }

  SPRINTF("   Host scsi%d: usb-storage\n", hostno);
  scsi_host_put(hostptr);
  return length;
}
```


A scsi driver: scsiglue.c (simplified proc_info function)

```
static int usb_storage_proc_info (
         char *buffer, char **start, off_t offset,
         int length, int hostno, int inout)
{
  struct us_data *us;
  struct Scsi_Host *hostptr;

  hostptr = scsi_host_hn_get(hostno);
  if (!hostptr) { return -ESRCH; }

  us = (struct us_data*)hostptr->hostdata[0];
  if (!us) {
    scsi_host_put(hostptr);
    return -ESRCH;
  }

  SPRINTF("   Host scsi%d: usb-storage\n", hostno);
  scsi_host_put(hostptr);
  return length;
}
```

A scsi driver: scsiglue.c (simplified proc_info function)

```
static int usb_storage_proc_info (struct Scsi_Host *hostptr,
         char *buffer, char **start, off_t offset,
         int length, int hostno, int inout)
{
  struct us_data *us;




  us = (struct us_data*)hostptr->hostdata[0];
  if (!us) {

    return -ESRCH;
  }

  SPRINTF("   Host scsi%d: usb-storage\n", hostno);

  return length;
}
```


scsiglue patch file

```
--- scsiglue_old.c      Tue Feb 13 13:31:56 2007
+++ scsiglue_new.c      Tue Feb 13 13:31:49 2007
@@ -1,20 +1,14 @@
-static int usb_storage_proc_info (
+static int usb_storage_proc_info (struct Scsi_Host *hostptr,
          char *buffer, char **start, off_t offset,
          int length, int hostno, int inout)
 {
   struct us_data *us;
-  struct Scsi_Host *hostptr;
-
-  hostptr = scsi_host_hn_get(hostno);
-  if (!hostptr) { return -ESRCH; }

   us = (struct us_data*)hostptr->hostdata[0];
   if (!us) {
-    scsi_host_put(hostptr);
     return -ESRCH;
   }

   SPRINTF("   Host scsi%d: usb-storage\n", hostno);
-  scsi_host_put(hostptr);
   return length;
}
```


Another scsi driver: sym53c8xx.c

```
static int sym53c8xx_proc_info(
                              char *buffer, char **start, off_t offset,
                              int length, int hostno, int func) {
     struct Scsi_Host *host;
     struct host_data *host_data;
     ncb_p ncb = 0;
     int retv;

     printk("sym53c8xx_proc_info: hostno=%d, func=%d\n", hostno, func);
     host = scsi_host_hn_get(hostno);
     if (!host) return -EINVAL;
     host_data = (struct host_data *) host->hostdata;
     ncb = host_data->ncb;
     retv = -EINVAL;
     if (!ncb) goto out;
     if (func) {
       retv = ncr_user_command(ncb, buffer, length);
     } else {
       if (start) *start = buffer;
       retv = ncr_host_info(ncb, buffer, offset, length);
     }
out: scsi_host_put(host);
     return retv;
}
```

Another scsi driver: sym53c8xx.c

```
static int sym53c8xx_proc_info(struct Scsi_Host *host,
                               char *buffer, char **start, off_t offset,
                               int length, int hostno, int func) {

     struct host_data *host_data;
     ncb_p ncb = 0;
     int retv;

     printk("sym53c8xx_proc_info: hostno=%d, func=%d\n", hostno, func);


     host_data = (struct host_data *) host->hostdata;
     ncb = host_data->ncb;
     retv = -EINVAL;
     if (!ncb) goto out;
     if (func) {
       retv = ncr_user_command(ncb, buffer, length);
     } else {
       if (start) *start = buffer;
       retv = ncr_host_info(ncb, buffer, offset, length);
     }
out:
     return retv;
}
```


sym53c8xx.c patch file

```
@@ -1,14 +1,11 @@
-static int sym53c8xx_proc_info(
+static int sym53c8xx_proc_info(struct Scsi_Host *host,
                               char *buffer, char **start, off_t offset,
                               int length, int hostno, int func) {
-  struct Scsi_Host *host;
   struct host_data *host_data;
   ncb_p ncb = 0;
   int retv;

   printk("sym53c8xx_proc_info: hostno=%d, func=%d\n", hostno, func);
-  host = scsi_host_hn_get(hostno);
-  if (!host) return -EINVAL;
   host_data = (struct host_data *) host->hostdata;
   ncb = host_data->ncb;
   retv = -EINVAL;
@@ -20,6 +17,5 @@
       *start = buffer;
     retv = ncr_host_info(ncb, buffer, offset, length);
   }
-out: scsi_host_put(host);
-  return retv;
+out: return retv;
 }
```


Observations

Code must be added and removed.

The context of the API usage can be affected as well.

Affected code may be scattered.

Need to be able to abstract over local variable names, device-specific computations, etc.

- `hostptr` vs. `host`, etc.

Need to be precise.

Comparing the two functions

```
static int usb_storage_proc_info (
 char *buffer,char **start,
 off_t offset,
 int length,int hostno,int inout) {
 struct us_data *us;
 struct Scsi_Host *hostptr;

 hostptr = scsi_host_hn_get(hostno);
 if (!hostptr) { return -ESRCH; }

 us=(struct us_data*)
       hostptr->hostdata[0];
 if (!us) {
   scsi_host_put(hostptr);
   return -ESRCH;
 }

 SPRINTF("...", hostno);
 scsi_host_put(hostptr);
 return length;
}
```

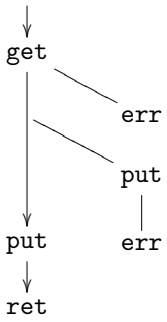
```
static int sym53c8xx_proc_info (
 char *buffer,char **start,
 off_t offset,
 int length,int hostno,int func) {
 struct Scsi_Host *host;
 struct host_data *host_data;
 ncb_p ncb = 0; int retv;

 printk("...", hostno, func);
 host = scsi_host_hn_get(hostno);
 if (!host) return -EINVAL;
 host_data =
  (struct host_data *) host->hostdata;
 ncb = host_data->ncb;
 retv = -EINVAL;
 if (!ncb) goto out;
 ...
out: scsi_host_put(host);
 return retv;
}
```

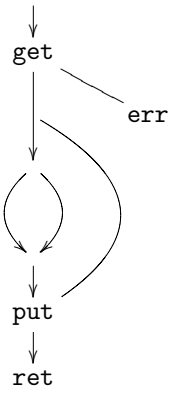
16

Comparing the control flow of the two functions

scsiglue.c



sym53c8xx.c



Each path does a get, then a put.

17

Observations

Code must be added and removed.

The context of the API usage can be affected as well.

Affected code may be scattered.

Need to be able to abstract over local variable names, device-specific computations, etc.

- `hostptr` vs. `host`, etc.

Need to be precise.

Need to describe control-flow paths, not abstract syntax trees.

18

Ideas

Follow the patch syntax!

- \- for removed code.
- \+ for added code.
- context code.

Metavariables and "..." for irrelevant code fragments.

At the statement level, "..." represents a control-flow path, not an abstract syntax tree.

SmPL: a language for writing semantic patches

```
  f();
  ...
- g();
+ h();
```

Creating a proc_info semantic patch

```
static int sym53c8xx_proc_info(

                              char *buffer, char **start, off_t offset,
                              int length, int hostno, int func) {
     struct Scsi_Host *host;
     struct host_data *host_data;
     ncb_p ncb = 0; int retv;

     printk("sym53c8xx_proc_info: hostno=%d, func=%d\n", hostno, func);
     host = scsi_host_hn_get(hostno);
     if (!host) return -EINVAL;
     host_data = (struct host_data *) host->hostdata;
     ncb = host_data->ncb;
     retv = -EINVAL;
     if (!ncb) goto out;
     if (func) { retv = ncr_user_command(ncb, buffer, length); }
     else { if (start) *start = buffer;
               retv = ncr_host_info(ncb, buffer, offset, length); }
out: scsi_host_put(host);
     return retv;
}
```


Creating a proc_info semantic patch

Drop the uninteresting parts

```
static int sym53c8xx_proc_info(

                              char *buffer, char **start, off_t offset,
                              int length, int hostno, int func) {
     struct Scsi_Host *host;
     ...


     host = scsi_host_hn_get(hostno);
     if (!host) return -EINVAL;
     ...




    scsi_host_put(host);
    ...
}
```

Creating a proc_info semantic patch

Indicate code to add and remove

```
static int sym53c8xx_proc_info(
+                               struct Scsi_Host *host,
                                char *buffer, char **start, off_t offset,
                                int length, int hostno, int func) {
-    struct Scsi_Host *host;
     ...


-    host = scsi_host_hn_get(hostno);
-    if (!host) return -EINVAL;
     ...




-    scsi_host_put(host);
     ...
}
```


Creating a proc_info semantic patch

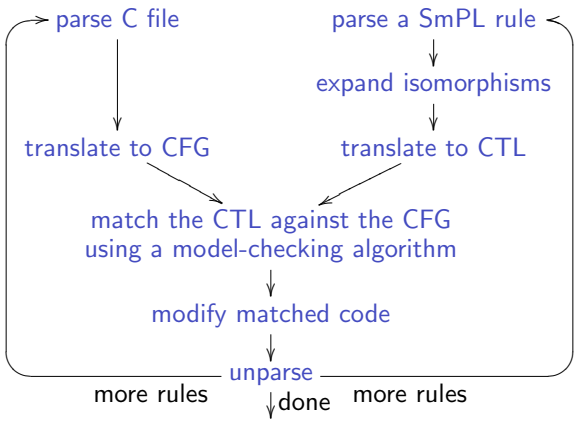
Abstract over variable names, arbitrary expressions, etc.

```
@@
identifier proc_info_fn;
identifier host, buffer, start, offset, length, hostno, func;
@@

static int proc_info_fn(
+                       struct Scsi_Host *host,
                        char *buffer, char **start, off_t offset,
                        int length, int hostno, int func) {
- struct Scsi_Host *host;
  ...
- host = scsi_host_hn_get(hostno);
- if (!host) return ...;
  ...
- scsi_host_put(host);
  ...
}
```

Function prototypes updated automatically.

Overview of the Coccinelle implementation



Implementation issues

Parse C file

- Maintain spacing, comments, preprocessor code.

Parse a SmPL rule

- Arbitrary interleaving of − and + code.
- Expansion according to isomorphisms:

```
X != NULL <=> NULL != X => X
```

Match and modify:

- A rule is matched against each function once (termination guaranteed).
- Transformation is only performed for a complete match.

The matching process

Ideas:

- Properties of paths are naturally expressed using temporal logic (CTL) [Lacey and de Moor].
- CTL has an efficient, easy-to-implement decision procedure: model checking.

```
@@ @@
  f();
  ...
- g();
+ h();
```

```
int main () {
  f();
  x();
  if (a < b) { g(); p(a); }
  else { p(b); g(); }
}
```

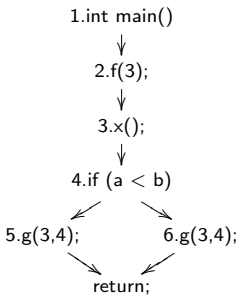
$f() \wedge AX(A[\neg(f() \vee g())\, U\, g()])$

Adding metavariables

Introduce predicates, with finite domain

```
@@
expression E1, E2;
@@

  f(E1);
  ...
- g(E1,E2);
+ h(E1,E2);
```



$f(E1) \wedge AX(A[\neg(f(E1) \vee g(E1,E2))\, U\, g(E1,E2)])$

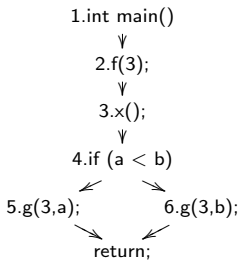
- $f(E1)$ matches at $(2, [E1 \mapsto 3])$
- $g(E1,E2)$ matches at $(5, [E1 \mapsto 3, E2 \mapsto 4])$, $(6, [E1 \mapsto 3, E2 \mapsto 4])$

These environments are compatible.
Negated bindings also allowed: constructive negation.

Finer-grained matching of metavariables

```
@@
expression E1, E2;
@@

  f(E1);
  ...
- g(E1,E2);
+ h(E1,E2);
```



$$f(E1) \wedge AX(A[\neg(f(E1) \vee g(E1,E2))\, U\, g(E1,E2)])$$

- $f(E1)$ matches at $(2, [E1 \mapsto 3])$
- $g(E1,E2)$ matches at
  $(5, [E1 \mapsto 3, E2 \mapsto a])$, $(6, [E1 \mapsto 3, E2 \mapsto b])$

These environments are not compatible.

Solution: Add quantifiers:

$$\exists E1.f(E1) \wedge AX(A[\neg(f(E1) \vee \exists E2.g(E1,E2))\, U\, \exists E2.g(E1,E2)])$$

Collecting the match information

Traditionally, model checking says yes or no

- CTL model checking internally collects all states where a formula holds.

We need to know:

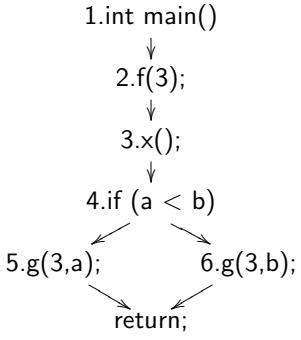
- Where to transform?
- With respect to what environment?

Solution:

- Collect witnesses for quantified metavariables.
- Introduce quantified metavariables for information we want to collect.

Matching with witnesses

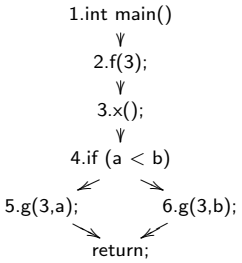
$$\exists E1.f(E1) \wedge AX(A[\neg(f(E1) \vee \exists E2.g(E1,E2))\, U\, \exists E2.g(E1,E2)])$$



- $f(E1)$ matches at $(2, [E1 \mapsto 3], \emptyset)$
- $g(E1,E2)$ matches at
  $(5, [E1 \mapsto 3, E2 \mapsto a], \emptyset)$,
  $(6, [E1 \mapsto 3, E2 \mapsto b], \emptyset)$
- $\exists E2.g(E1,E2)$ matches at
  $(5, [E1 \mapsto 3], \{(5, [E2 \mapsto a], \emptyset)\})$,
  $(6, [E1 \mapsto 3], \{(6, [E2 \mapsto b], \emptyset)\})$

Collecting extra information

$\exists E1.\mathtt{f}(E1) \wedge \mathsf{AX}(\mathsf{A}[\neg(\mathtt{f}(E1) \vee \exists E2.\mathtt{g}(E1,E2)) \,\mathsf{U}\, \exists E2.\exists v.\mathtt{g}(E1,E2)^{v}])$



- $\mathtt{f}(E1)$ matches at $(2, [E1 \mapsto 3], \emptyset)$
- $\exists E2.\exists v.\mathtt{g}(E1,E2)^{v}$ matches at
  $(5,[E1 \mapsto 3],$
  $\{(5,[E2 \mapsto \mathtt{a}],$
  $\{(5,[v \mapsto \mathtt{g}(E1,E2)],\emptyset)\})),$
  $(6,\ldots,\ldots)$

Final answer:

$(2,[],\{(2,[E1 \mapsto 3],\{(5,[E2 \mapsto \mathtt{a}],\{(5,[v \mapsto \mathtt{g}(E1,E2)],\emptyset)\}),(6,\ldots)\})\})$

Interpretation:

- Transform at node 5 according to the transformation associated with $\mathtt{g}(E1,E2)$, where $E1$ is 3 and $E2$ is a.
- Similarly for node 6.

Benefits of our approach

CTL algorithm is easy to implement.

- Efficient enough for individual driver functions.

Flexible logic encoding:

- (Mostly) universal path quantification for transformation.
- Existential path quantification for searching.
- Either encoding can be extended to collect partial matches.

Current status

- SmPL compiler and CTL algorithm implemented.
- Semantic patches written for over 70 Linux collateral evolutions.
- Semantic patches applied to over 6000 relevant driver files, from recent Linux versions.
  - Often less than one second per relevant file.
- Some patches contributed to Linux (reaction: "Cool", "Great").
- Soundness and completeness proofs well underway.

Conclusion

- Transformation of multiple program points related by control-flow relationships is interesting in practice.
- CTL with some extensions is a convenient target language for expressing such transformations.
- CTL model checking is efficient enough for performing the matching required by such transformations in practice.
- Perhaps our approach is not restricted to device drivers or to collateral evolutions, but we make no promises...

34

35

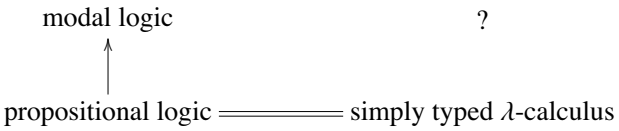
Relationship between Single and Multi Context Formulations of Modal Calculi

Tatsuya Abe
CVS, AIST
(IST until September 30, 2007)

Third DIKU-IST
October 5, 2007

Motivation for Construction of Modal Calculi (1/2)

Fact. 1. Modal logic is an *extension* of propositional logic.
2. Simply typed λ-calculus corresponds to (intuitionistic) propositional logic via *Curry-Howard isomorphism*.



In this study, we construct λ-calculi corresponding to modal logic via Curry-Howard isomorphism for modal logic.

Motivation for Construction of Modal Calculi (2/2)

Why do we want to construct typed λ-calculi corresponding to modal logic?

By extending logic and Curry-Howard isomorphism, we have obtained new typed λ-calculi and new types.

E.g. 1. second-order propositional logic, $\lambda 2$ (system F), polymorphic types
2. first-order predicate logic, λP, dependent types

Similarly, we expect to find new typed λ-calculi (called *modal calculi*) and a new kind of type (called *modal types*).

Brief History of Modal Calculi

Martini and Masini	1994	IK	
Pfenning and Wong	1995	IS4	
Bierman and de Paiva	2000	IS4	λ^{S4}
Alechina, Mendler, de Paiva, and Ritter	2001	IS4	
Bellin, de Paiva, and Ritter	2001	IK	
Pfenning and Davies	2001	IS4	$\lambda_e^{\to\Box}$
Kakutani and Abe	2007	IK	

We will refer to λ^{S4} and $\lambda_e^{\to\Box}$ later.

Modal Logic

The set of types in modal propositional logic is as follows,

$$A ::= p \mid A \supset A \mid \Box A$$

where p ranges over the set of propositional variables.

A set of types is written as $\Gamma, \Delta, \ldots$.

A set of types Γ is said to be a *context*.

A *judgment* is written as $\Gamma \vdash A$.

$\Gamma \vdash A$ means that A is true under $\bigwedge\{ B \mid B \in \Gamma \}$.
$\Gamma \vdash \Box A$ means that A is *necessarily* true under $\bigwedge\{ B \mid B \in \Gamma \}$.

Context Formulations

We focus on difference in context formulations.

Usually, judgments are of the form $\Gamma \vdash A$.

Judgments of the form $\Delta \mid \Gamma \vdash A$ are also used.

$\Delta \mid \Gamma \vdash A$ denotes that A is true under

$$\bigwedge\{ \Box B \mid B \in \Delta \} \wedge \bigwedge\{ C \mid C \in \Gamma \} \ .$$

Terminology.	$\Gamma \vdash A$	single context formulation
	$\Delta \mid \Gamma \vdash A$	multi context formulation

Modal Calculus of Single Contexts

λ^{SS4}: IS4, single contexts, similar to Bierman and de Paiva's λ^{S4}

$$\Gamma, x\colon A \vdash x\colon A$$

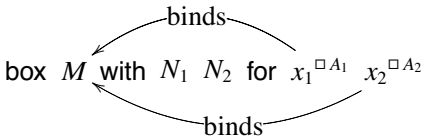
$$\frac{\Gamma, x\colon A \vdash M\colon B}{\Gamma \vdash \lambda x^{A}.M\colon A \supset B} \qquad \frac{\Gamma \vdash M\colon A \supset B \qquad \Gamma \vdash N\colon A}{\Gamma \vdash MN\colon B}$$

$$\frac{\Gamma \vdash N_i\colon \Box A_i\ (1 \le i \le n) \qquad x_1\colon \Box A_1, \ldots, x_n\colon \Box A_n \vdash M\colon B}{\Gamma \vdash \mathsf{box}\ M\ \mathsf{with}\ N_1 \cdots N_n\ \mathsf{for}\ x_1{}^{\Box A_1} \cdots x_n{}^{\Box A_n}\colon \Box B}$$

$$\frac{\Gamma \vdash M\colon \Box A}{\Gamma \vdash \mathsf{unbox}(M)\colon A}$$

Multi (Simultaneous) Bindings in λ^{SS4}

$$\frac{\Gamma \vdash N_i\colon \Box A_i\ (1 \le i \le n) \qquad x_1\colon \Box A_1, \ldots, x_n\colon \Box A_n \vdash M\colon B}{\Gamma \vdash \mathsf{box}\ M\ \mathsf{with}\ N_1 \cdots N_n\ \mathsf{for}\ x_1{}^{\Box A_1} \cdots x_n{}^{\Box A_n}\colon \Box B}$$



Such multi (simultaneous) bindings are *not popular* in the theory of λ-calculi.

Modal Calculus of Multi Contexts

λ^{MS4}: IS4, multi contexts, similar to Pfenning and Davies's $\lambda_e^{\to\Box}$

$$\Delta \mid \Gamma, x\colon A \vdash x\colon A \qquad \Delta, x\colon A \mid \Gamma \vdash x\colon A$$

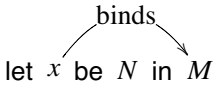
$$\frac{\Delta \mid \Gamma, x\colon A \vdash M\colon B}{\Delta \mid \Gamma \vdash \lambda x^{A}.M\colon A \supset B} \qquad \frac{\Delta \mid \Gamma \vdash M\colon A \supset B \qquad \Delta \mid \Gamma \vdash N\colon A}{\Delta \mid \Gamma \vdash MN\colon B}$$

$$\frac{\Delta \mid \emptyset \vdash M\colon A}{\Delta \mid \Gamma \vdash \Box M\colon \Box A} \qquad \frac{\Delta \mid \Gamma \vdash N\colon \Box A \qquad \Delta, x\colon A \mid \Gamma \vdash M\colon B}{\Delta \mid \Gamma \vdash \mathsf{let}\,\Box\, x^{A}\ \mathsf{be}\ N\ \mathsf{in}\ M\colon B}$$

Proposition. *λ^{SS4} and λ^{MS4} are equivalent w.r.t. derivability of types, i.e., $\exists M.\, \Box\Delta, \Gamma \vdash M\colon A$ if and only if $\exists M.\, \Delta \mid \Gamma \vdash M\colon A$.*

Let-terms in λ^{MS4}

A λ-term of the form



is adopted in λ^{MS4}. Such single bindings are *popular* in the theory of λ-calculi (e.g., ML, Scheme).

Summary

	context	binding
λ^{SS4} and λ^{S4}	single (popular)	multi (not popular)
λ^{MS4} and $\lambda_e^{\to\Box}$	multi (not popular)	single (popular)

Natural Question for Construction of Modal Calculi

Question. Which context formulation should we choose in constructing modal calculi?

Answer. We can choose whichever we like. It makes no essential difference.

When we construct a modal calculus (e.g., λ^{SS4}) based on single context formulation, we can also construct another modal calculus (e.g., λ^{MS4}) based on multi context formulation, and vice versa (e.g., we can construct λ^{MS4} from λ^{SS4}.)

Main Result in This Talk (1/4)

Derivability of types (provability) of calculi corresponding to IS4 are the same by definition (i.e., correspondence to IS4).

How about relationship between λ^{SS4}*-terms* and λ^{MS4}*-terms*?

There exists a one-to-one correspondence between terms. Formally,

Theorem. *There exist functions* $(\cdot)^\bullet$ *and* $(\cdot)^\circ$ *such that*

1. *if* $\Gamma \vdash_{\lambda^{\mathrm{SS4}}} M : A$, *then* $\emptyset \mid \Gamma \vdash_{\lambda^{\mathrm{MS4}}} M^\bullet : A$.
2. *If* $\Delta \mid \Gamma \vdash_{\lambda^{\mathrm{MS4}}} M : A$, *then* $\Box\Delta, \Gamma \vdash_{\lambda^{\mathrm{SS4}}} M^\circ : A$.

We define a function $(\cdot)^\bullet$ from λ^{SS4} to λ^{MS4} as follows,

$$(\mathsf{box}\ M\ \mathsf{with}\ \overrightarrow{N}\ \mathsf{for}\ \overrightarrow{x^{\Box A}})^\bullet = \mathsf{let}\ \Box \overrightarrow{y^A}\ \mathsf{be}\ \overrightarrow{N^\bullet}\ \mathsf{in}\ \Box \overrightarrow{[\Box y/x]M^\bullet} \qquad \text{where } \overrightarrow{y}\text{s are fresh}$$

$$(\mathsf{unbox}(M))^\bullet = \mathsf{let}\ \Box x^A\ \mathsf{be}\ M^\bullet\ \mathsf{in}\ x \qquad \text{where } x \text{ is fresh}$$

where $\mathsf{let}\ \Box \overrightarrow{y^A}\ \mathsf{be}\ \overrightarrow{N}\ \mathsf{in}\ M$ denotes $\mathsf{let}\ \Box {y_1}^{A_1}\ \mathsf{be}\ N_1\ \mathsf{in} \cdots \mathsf{let}\ \Box {y_n}^{A_n}\ \mathsf{be}\ N_n\ \mathsf{in}\ M$. We define a function $(\cdot)^\circ$ from λ^{MS4} to λ^{SS4}. In order to emphasize that we define $(\cdot)^\circ$ on λ^{MS4}-judgments, we index λ^{MS4}-terms by their multi contexts (written as $M\ \mathit{under}\ \Delta \mid \Gamma$):

$$\begin{aligned}
(x\ \mathit{under}\ \Delta \mid \Gamma, x\colon A)^\circ &= x \\
(x\ \mathit{under}\ \Delta, x\colon A \mid \Gamma)^\circ &= \mathsf{unbox}(x) \\
(\lambda x^A.M\ \mathit{under}\ \Delta \mid \Gamma)^\circ &= \lambda x^A.(M\ \mathit{under}\ \Delta \mid \Gamma, x\colon A)^\circ \\
(MN\ \mathit{under}\ \Delta \mid \Gamma)^\circ &= (M\ \mathit{under}\ \Delta \mid \Gamma)^\circ (N\ \mathit{under}\ \Delta \mid \Gamma)^\circ \\
(\Box M\ \mathit{under}\ \overrightarrow{z\colon B} \mid \Gamma)^\circ &= \mathsf{box}\ \overrightarrow{[y/z]}(M\ \mathit{under}\ \overrightarrow{z\colon B} \mid \emptyset)^\circ\ \mathsf{with}\ \overrightarrow{z}\ \mathsf{for}\ \overrightarrow{y^{\Box B}} \\
(\mathsf{let}\ \Box x^A\ \mathsf{be}\ N\ \mathsf{in}\ M\colon B\ \mathit{under}\ \Delta \mid \Gamma)^\circ &= [(N\ \mathit{under}\ \Delta \mid \Gamma)^\circ/x](M\ \mathit{under}\ \Delta, x\colon A \mid \Gamma)^\circ\ .
\end{aligned}$$

where $\overrightarrow{y}$s are fresh. In the following, $(M\ \mathit{under}\ \Delta \mid \Gamma)^\circ$ is simply written as M°.

Main Result in This Talk (2/4)

The equations between λ^{SS4}-terms are

$$\begin{aligned}
&(\lambda x.M)N = [N/x]M \\
&\lambda x.Mx = M && \text{if } x \notin \mathsf{fv}\ M \\
&\mathsf{unbox}(\mathsf{box}\ M\ \mathsf{with}\ \overrightarrow{N}\ \mathsf{for}\ \overrightarrow{x}) = \overrightarrow{[N/x]}M \\
&\mathsf{box}\ \mathsf{unbox}(M)\ \mathsf{with}\ \overrightarrow{N}\ \mathsf{for}\ \overrightarrow{x} = \overrightarrow{[N/x]}M \\
&\mathsf{box}\ M\ \mathsf{with}\ N_1, \ldots, N_{i-1}, N_i, N_{i+1}, \ldots, N_n\ \mathsf{for}\ x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n \\
&= \mathsf{box}\ M\ \mathsf{with}\ N_1, \ldots, N_{i-1}, N_{i+1}, \ldots, N_n\ \mathsf{for}\ x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n && \text{if } x_i \notin \mathsf{fv}\ M \\
&\mathsf{box}\ M\ \mathsf{with}\ N_1, \ldots, N_{j-1}, N_i, N_{j+1}, \ldots, N_n\ \mathsf{for}\ x_1, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_n \\
&= \mathsf{box}[x_i/x_j]M\ \mathsf{with}\ N_1, \ldots, N_{j-1}, N_{j+1}, \ldots, N_n\ \mathsf{for}\ x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n \\
&\mathsf{box}\ M\ \mathsf{with}\ N_1, \ldots, N_i, \ldots, N_j, \ldots, N_n\ \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n \\
&= \mathsf{box}\ M\ \mathsf{with}\ N_1, \ldots, N_j, \ldots, N_i, \ldots, N_n\ \mathsf{for}\ x_1, \ldots, x_j, \ldots, x_i, \ldots, x_n\ .
\end{aligned}$$

Main Result in This Talk (3/4)

The equations of λ^{MS4}-terms are

$$\begin{aligned}
(\lambda x.M)N &= [N/x]M \\
\lambda x.Mx &= M && \text{if } x \notin \mathsf{fv}\ M \\
\mathsf{let}\ \Box x\ \mathsf{be}\ \Box N\ \mathsf{in}\ M &= [N/x]M \\
\mathsf{let}\ \Box x\ \mathsf{be}\ M\ \mathsf{in}\ \Box x &= M \\
L(\mathsf{let}\ \Box x\ \mathsf{be}\ N\ \mathsf{in}\ M) &= \mathsf{let}\ \Box x\ \mathsf{be}\ N\ \mathsf{in}\ LM && \text{if } x \notin \mathsf{fv}\ L \\
(\mathsf{let}\ \Box x\ \mathsf{be}\ N\ \mathsf{in}\ M)L &= \mathsf{let}\ \Box x\ \mathsf{be}\ N\ \mathsf{in}\ ML && \text{if } x \notin \mathsf{fv}\ L \\
\lambda y.\ \mathsf{let}\ \Box x\ \mathsf{be}\ N\ \mathsf{in}\ M &= \mathsf{let}\ \Box x\ \mathsf{be}\ N\ \mathsf{in}\ \lambda y.M && \text{if } y \notin \mathsf{fv}\ N \\
\mathsf{let}\ \Box x\ \mathsf{be}\ \mathsf{let}\ \Box y\ \mathsf{be}\ N\ \mathsf{in}\ M\ \mathsf{in}\ L &= \mathsf{let}\ \Box y\ \mathsf{be}\ N\ \mathsf{in}\ \mathsf{let}\ \Box x\ \mathsf{be}\ M\ \mathsf{in}\ L && \text{if } y \notin \mathsf{fv}\ L \\
\Box\ \mathsf{let}\ \Box x\ \mathsf{be}\ N\ \mathsf{in}\ M &= \mathsf{let}\ \Box x\ \mathsf{be}\ N\ \mathsf{in}\ \Box M \\
\mathsf{let}\ \Box y\ \mathsf{be}\ N\ \mathsf{in}\ \mathsf{let}\ \Box x\ \mathsf{be}\ M\ \mathsf{in}\ L &= \mathsf{let}\ \Box x\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ \Box y\ \mathsf{be}\ N\ \mathsf{in}\ L && \text{if } x \notin \mathsf{fv}\ N \text{ and } y \notin \mathsf{fv}\ M.
\end{aligned}$$

Main Result in This Talk (4/4)

The correspondence preserves *equations* of λ^{SS4}-terms conservatively. Formally,

Theorem. *1. Assume* $\Gamma \vdash_{\lambda^{\mathrm{SS4}}} M : A$, $\Gamma \vdash_{\lambda^{\mathrm{SS4}}} N : A$. *Then,* $M =_{\lambda^{\mathrm{SS4}}} N$ *if and only if* $M^\bullet =_{\lambda^{\mathrm{MS4}}} N^\bullet$.

2. Assume $\Delta \mid \Gamma \vdash_{\lambda^{\mathrm{MS4}}} M : C$, $\Delta \mid \Gamma \vdash_{\lambda^{\mathrm{MS4}}} N : C$. *Then,* $M =_{\lambda^{\mathrm{MS4}}} N$ *if and only if* $M^\circ =_{\lambda^{\mathrm{SS4}}} N^\circ$.

Contribution (1/3)

1. We gave translations between two modal calculi based single and multi context formulations.
2. We found appropriate equations for IS4 (→ next slide).

Contribution (2/3)

λ^{S4}: Bierman and de Paiva's calculus

$$\text{the set of } \lambda^{\mathrm{SS4}}\text{-terms} = \text{the set of } \lambda^{\mathrm{S4}}\text{-terms}$$
$$\text{the set of } \lambda^{\mathrm{SS4}}\text{-equations} \supseteq \text{the set of } \lambda^{\mathrm{S4}}\text{-equations}$$

$\lambda_e^{\to\Box}$: Pfenning and Davies's calculus

$$\text{the set of } \lambda^{\mathrm{MS4}}\text{-terms} = \text{the set of } \lambda_e^{\to\Box}\text{-terms}$$
$$\text{the set of } \lambda^{\mathrm{MS4}}\text{-equations} \supseteq \text{the set of } \lambda_e^{\to\Box}\text{-equations}$$

Contribution (3/3)

The calculi λ^{S4} and $\lambda_e^{\to\Box}$ do not have enough equations since their calculi do not have one-to-one correspondence.

On the other hand, our calculi λ^{SS4} and λ^{MS4} have one-to-one correspondence since our calculi have *appropriate* equations for IS4.

Contributions of this study:

1. We gave translations between two modal calculi based on single and multi context formulations.
2. We found appropriate equations for IS4.

Future Work: On Other Modalities

Why have we discussed IS4? On other modalities (e.g., IK)?

In IS4, $\Box A \supset A$ and $\Box A \supset \Box\Box A$ are admissible while $A \supset \Box A$ is not admissible.

Therefore, $\Box A$ and $\Box \cdots \Box A$ are logically equivalent while A and $\Box A$ are not.

For IS4, contexts are divided into *two*, i.e., $\Delta \mid \Gamma \vdash M : A$.

For IK, we must adopt $\Delta_n \mid \cdots \mid \Delta_1 \mid \Gamma \vdash M : A$ since A, $\Box A$, $\Box\Box A$, ... are not equivalent in IK. Such a calculus seems complex and difficult to handle. We dealt with IS4 only, for simplicity. The question on IK (and so on) is left open.

ArchX: A Synchronization Framework for Tree-Structured Data

Izumi MIHASHI
Information Processing Laboratory
The University of Tokyo

Synchronization of XML Documents Stored in Different Formats

- Bookmarks of web browsers
- Calendars used on PC and PDA
- Address books

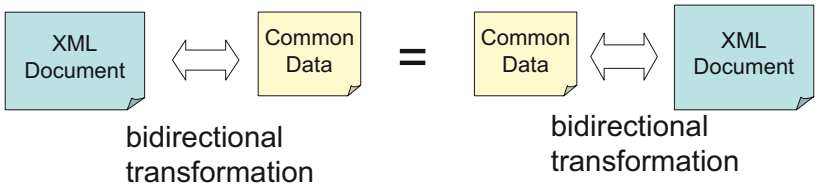
```
<lists>
  <note>Address Book A</note>
  <item><n>Ginger</n>
    <mail>mgm.com</mail>
    <mail>mac.com</mail>
  </item>
  <item><n>John</n></item>
</lists>
```

```
<book>
  <note>Address Book B</note>
  <entry>Ginger
    <adds>
      <add>mgm.com</add>
      <add>mac.com</add>
    </adds>
  </entry>
  <entry>John<adds></adds></entry>
</book>
```

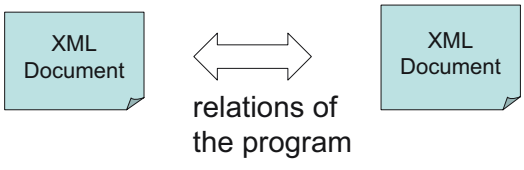
1

How to define the status that documents are synchronized

- Harmony



- ArchX



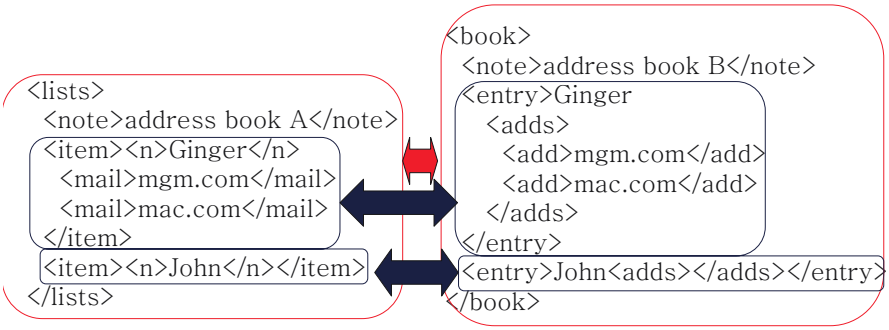
2

Outline

- Introduction
- **Description of the Correspondence between Documents**
- Demo
- Formal Definition of the Synchronization
- Synchronization Algorithm
- Conclusion and Future Work

3

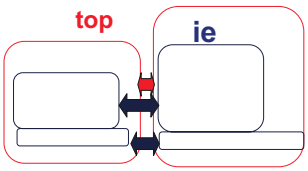
Description of the Correspondence between Documents



4

Description of the Correspondence between Documents

```
relation top =
  lists[note[String], (var item)*]
    <->
  book[note[String], (var entry)*]
    where ie(item, entry)
```



top

```
relation ie =
  item[n[var nm as String],
       (mail[var ml as String])*]
  <->
  entry[var nm as String,
        adds[(add[var ml as String])*]]
```

ie

From biXid[Kawanaka and Hosoya 2007]

5

Outline

- Introduction
- Description of the Correspondence between Formats
- **Demo**
- Formal Definition of the Synchronization
- Synchronization Algorithm
- Conclusion and Future Work

6

Outline

- Introduction
- Description of the Correspondence between Formats
- Demo
- **Formal Definition of the Synchronization**
  - Binding Tree
  - Definition of the Synchronization
- Synchronization Algorithm
- Conclusion and Future Work

7

Binding Tree

```
<lists>
  <note>address book A</note>
  <item><n>Ginger</n>
    <mail>mgm.com</mail>
    <mail>mac.com</mail>
  </item>
  <item><n>John</n></item>
</lists>
```

Address Book A

```
item->
 [{nm->["Ginger"], ml->["mgm.com","mac.com"]},
  {nm->["John"], ml -> []}]
```

Binding Tree of Address Book A

8

Equivalent Binding Trees

where ie(item, entry)


```
item ->
   [{nm -> ["Ginger"],
     ml -> ["mgm.com",
            "mac.com"]},
    {nm -> ["John"],
     ml -> []}]
```

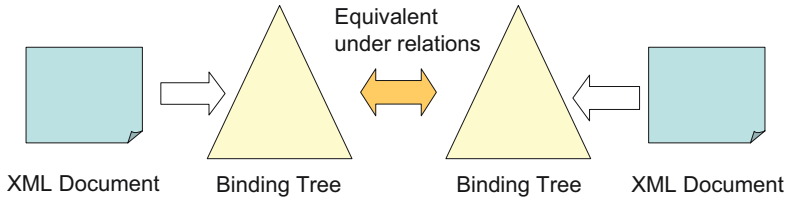
Binding Tree of Address Book A

```
entry ->
   [{nm -> ["Ginger"],
     ml -> ["mgm.com",
            "mac.com"]},
    {nm -> ["John"],
     ml -> []}]
```

Binding Tree of Address Book B

9

Definition of the Status that Documents are Synchronized




```
<lists>
  <note>
   address book A
  </note>
  ...
  <item>
   <n>John</n>
  </item>
</lists>
```

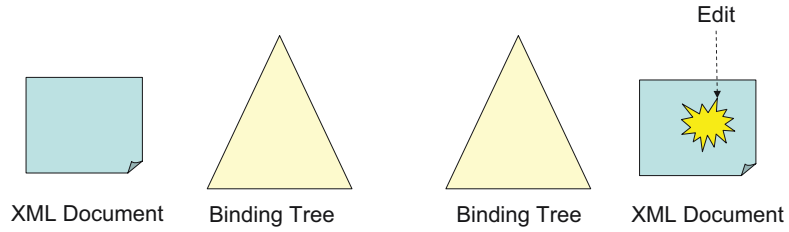
```
item ->
   [{nm -> ...,
     ml ->...},
    {nm -> John,
     ml -> ...}]
```

```
entry ->
   [{nm -> ...,
     ml ->...},
    {nm -> John
     ml -> ...}]
```

```
<book>
  <note>
    Address Book B
  </note>
  ...
<entry>
John
<adds></adds>
  </entry>
</book>
```

10

Required Property of Synchronized Documents




```
<lists>
  <note>
   address book A
  </note>
  ...
  <item>
   <n>John</n>
  </item>
</lists>
```

```
item ->
   [{nm -> ...,
     ml ->...},
    {nm -> John,
     ml -> ...}]
```

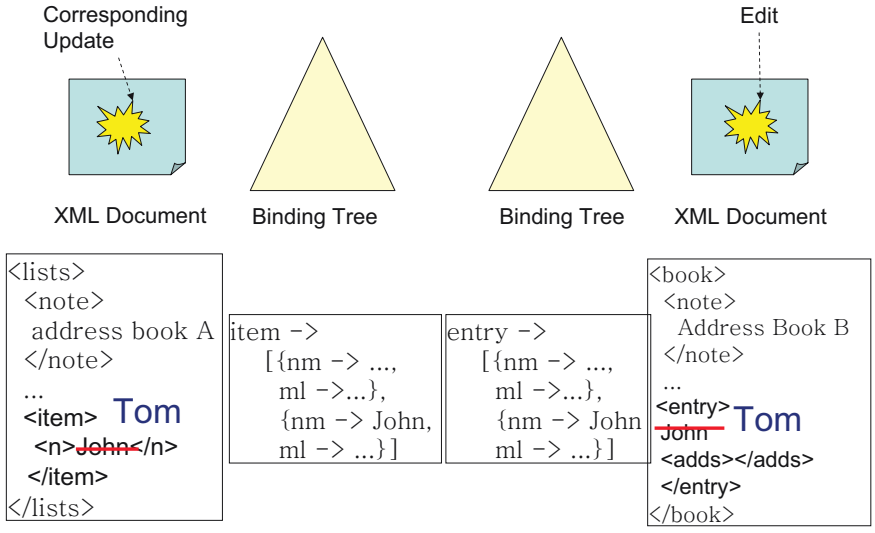
```
entry ->
   [{nm -> ...,
     ml ->...},
    {nm -> John
     ml -> ...}]
```

```
<book>
  <note>
    Address Book B
  </note>
  ...
<entry>
John
<adds></adds>
  </entry>
</book>
```

Tom

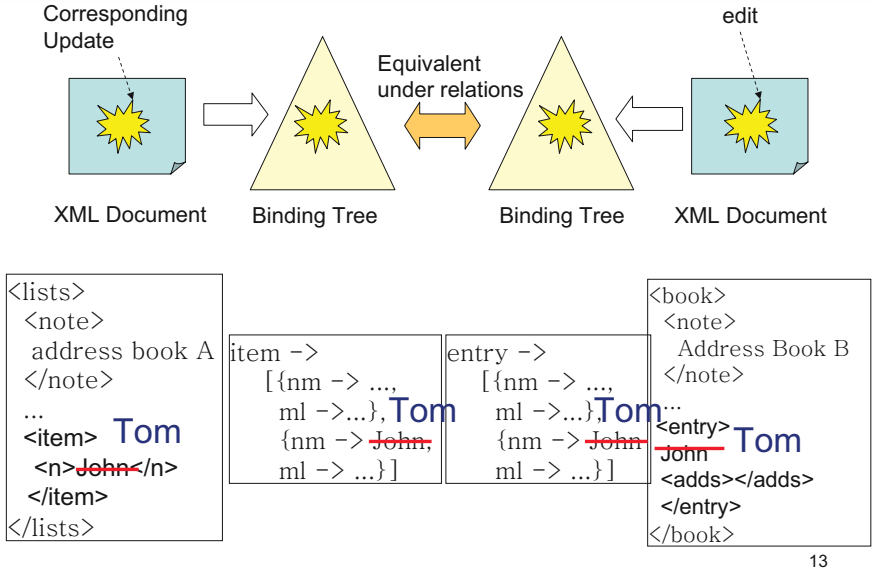
11

Required Property of Synchronized Documents



12

Required Property of Synchronization



13

Outline

- Introduction
- Description of the Correspondence between Formats
- Demo
- Formal Definition of the Synchronization
- **Synchronization Algorithm**
  - Marked Tree
  - Overview of Synchronization Algorithm
- Conclusion and Future Work

14

Marked Tree

- A data structure that shows how an XML document is matched to patterns.
- A number attached to a node of a marked tree shows which symbol it corresponds to.

```
<item>
  <n>Ginger</n>
  <mail>mgm.com</mail>
  <mail>mac.com</mail>
</item>
```

```
relation item =
  item5[n6[var nm7 as String8],
        (mail9[var ml10 as String11])*]
    <->
  ...
```

```
item5[
   n6[nm7{Ginger8}]
   mail9[ml10{mgm.com11}]
   mail9[ml10{mac.com11}]]}
```

15

Restoring a XML Document from a Marked Tree

The XML Document is obtained by removing numbers and variables.

```
lists1[note2[address book  A3]
  litem4{item5[n6[nm7{Ginger8}]...]}
  litem4{item5[n6[nm7{John8}]]}]
```

```
<lists>
   <note>address book A</note>
   <item><n>Ginger</n>...</item>
   <item><n>John</n></item>
</lists>
```

16

Restoring a Binding Tree from a Marked Tree

The Binding tree is obtained by binding the part enclosed by braces to variables.

```
lists1[note2[address book  A3]
  litem4{item5[n6[nm7{Ginger8}]...]}
  litem4{item5[n6[nm7{John8}]]}]
```

```
litem ->
   [{nm -> [“Ginger”],  ml -> ...},
    {nm -> [“John”], ml -> []}]
```

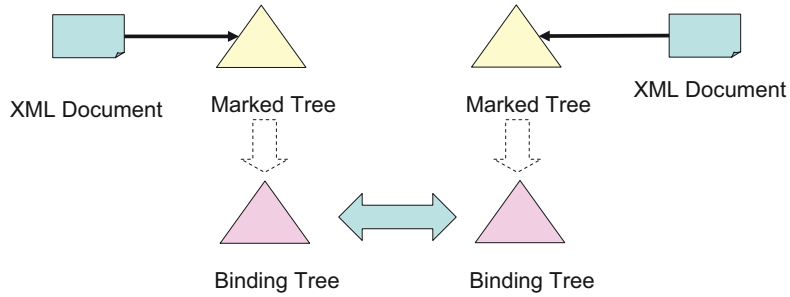
17

Overview of Synchronization Algorithm

1. The original documents

18

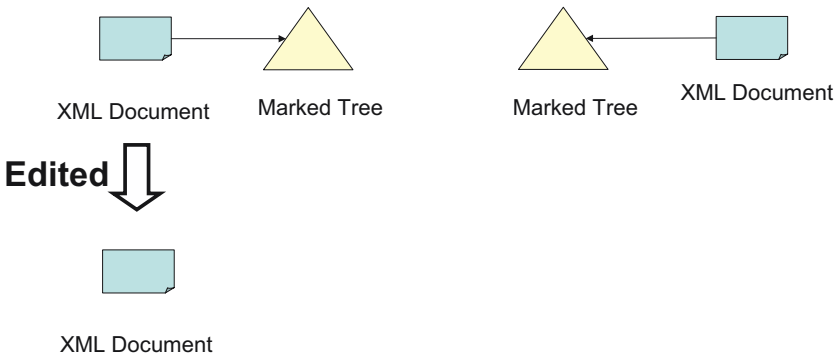
Overview of Synchronization Algorithm



2. Checks if the documents are synchronized

19

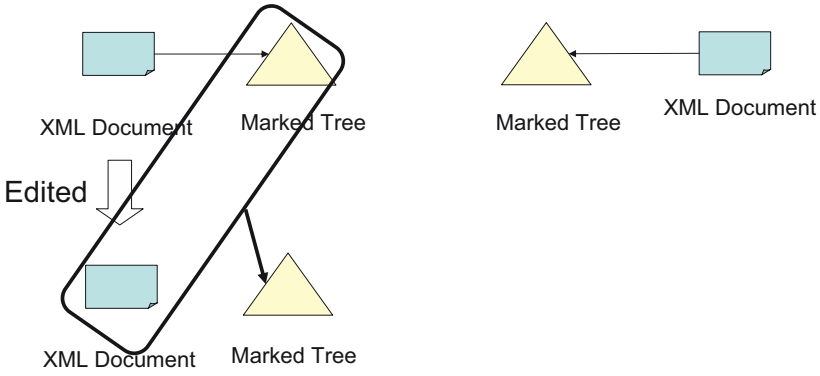
Overview of Synchronization Algorithm



3. One of the original documents is edited

20

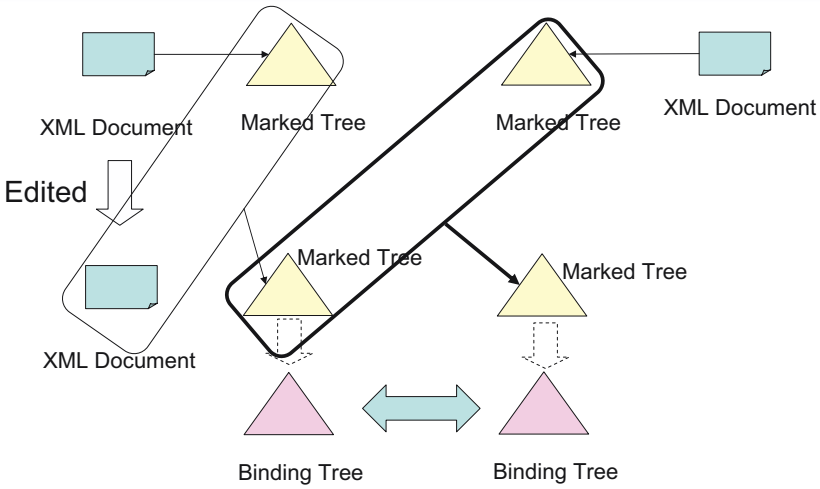
Overview of Synchronization Algorithm



4. Propagate the Edits to Marked Trees

21

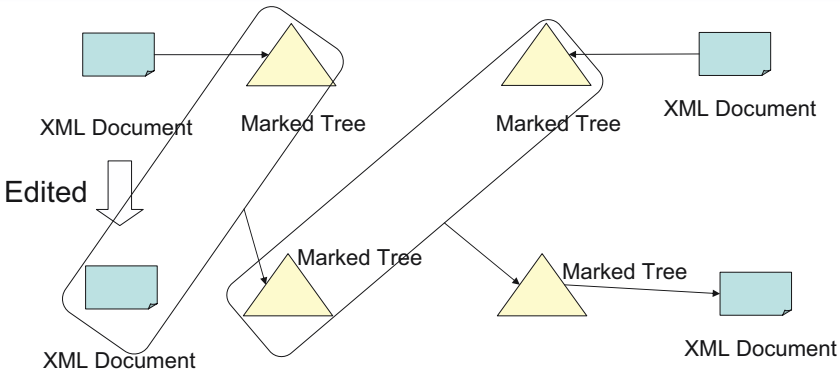
Overview of Synchronization Algorithm



5. Propagate the Edits to the Other Side

22

Overview of Synchronization Algorithm



6. Propagate Edits to the Other XML Document.

23

Outline

- Introduction
- Description of the Correspondence between Formats
- Demo

- Synchronization Algorithm
- **Conclusion and Future Work**

24

Conclusion and Future Work

We have designed a new framework for synchronizing XML documents in different formats called ArchX.

Future Work

- More than Two Documents
- Edits in Multiple Documents

25

Reversible Machine Code and Its Abstract Processor Architecture *Extended Abstract*⋆

Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama

DIKU, Department of Computer Science,
University of Copenhagen, DK-2100 Copenhagen, Denmark,

1 Introduction

This paper presents the principles behind a reversible processor architecture. We are interested in a reversible version of the *von Neumann architecture*, a classic computer design for sequential computation with a single processing unit and random access memory, instead of more theoretical models, such as Turing machines. One of the first reversible programmable processors fabricated is of this type [15, 16, 7]. We shall define an abstract machine and prove that all instruction sets for this machine that satisfy certain formal conditions, identified and presented in this paper, are reversible; *i.e.*, their semantics are forward and backward deterministic. A unique feature of the reversible abstract machine is a control logic that allows to change the direction of execution by flipping the direction bit. It is noteworthy that any program written in reversible machine code is guaranteed to be reversible; no programming error can break the reversibility.

The purpose of this work is to provide a clear specification of the interplay between the physical and software levels. Understanding this interface is important, as a challenge of reversible computing is that the *entire* computing system must be reversible, from the physical bottom to the abstract top.

We believe that reversible computation models have properties that are noteworthy in their own right (*e.g.*, forward and backward computation takes the same number of execution steps) and have interesting applications in other areas. For example, unconventional physical computation models, such as quantum computing, require that all computations are organized reversibly [6].

2 Reversible Abstract Machine

The design of the control logic is based on work by Vieri [15], Frank [7], Hall [11], and Cezzar [4]. Our contribution is a clear formalization of a reversible abstract machine for which we prove that any instruction set that satisfies certain conditions is reversible. A standard abstract machine performs *one-directional* execution of machine code, while a reversible abstract machine allows *bidirectional* (forward and backward) execution of reversible machine code.

⋆ Extended abstract of publication [2].

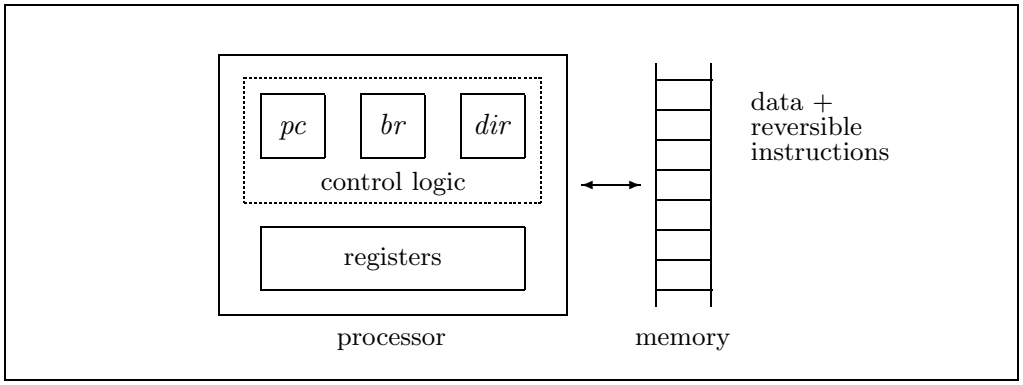


Fig. 1. Reversible abstract machine

The challenge of reversibility for an abstract machine is two-fold. First, the *instruction execution* must be reversible. This places restrictions on the instructions possible in such a reversible architecture. Specifically, they must be injective when considered as functions on machine states. Second, the *control logic* must be reversible. With control logic, we refer to the part of the processing unit controlling the *program counter*, including any interaction from the instructions. To ensure reversibility, this must also be an injective function on machine states.

Control Logic. The most difficult of these challenges is good design for control. At any point of program execution, the computation direction of a reversible machine can be switched (forward, backward). In a standard machine model, we constantly face the *orthogonality problem*. In general, we cannot know whether we have arrived at the current state by a jump or by sequential execution. Thus, we cannot reverse program execution and return to exactly the state that led to the current state. A classic solution to this problem is the generation of a *trace* [12, 3, 4]. For practical purposes, however, this is unsatisfactory, as the trace grows proportionally to the length of the computation.

A good design for a reversible processor has to take a *trace-free approach*. This can be accomplished in an architecture that has *three special-purpose registers* involved in the control logic (Fig. 1):

- A *program counter* (*pc*) for pointing to the current instruction.
- A *branch register* (*br*) for jumps.
- A *direction bit* (*dir*) for specifying execution direction.

Between instruction executions, we update the control state by the following two rules: (1) If the branch register is *zero*, add the direction bit to the program counter. The direction bit has the value 1 or -1, and thus the program counter is either incremented or decremented. This achieves sequential forward or backward execution of a program. (2) If the branch register is *non-zero*, add that (times the direction bit) to the program counter. A non-zero branch register indicates that a jump is to be performed. Instructions that require jumps will thus modify

the branch register and *not* the program counter directly. The effect of this is to make the control logic reversible by solving the orthogonality problem: the branch register is preserved after modifying the program counter. This is sufficient to determine where a jump came from. We will come back to this important point after formalizing the state model and the execution semantics.

State Model. We model the set of machine states, $\Sigma = \mathcal{M} \times \mathcal{R} \times \mathcal{C}$, as follows. A machine state $\sigma \in \Sigma$ is a triple $\sigma = (M, R, C)$ such that

$$\begin{array}{ll} \text{Memory:} & \mathcal{M} \ni M : \mathbb{Z}_{32} \to \mathbb{Z}_{32} \\ \text{Registers:} & \mathcal{R} \ni R : \mathit{RegNames} \to \mathbb{Z}_{32} \\ \text{Control:} & \mathcal{C} \ni C : \mathbb{Z}_{32} \times \mathbb{Z}_{32} \times \{1, -1\} \end{array}$$

where $\mathbb{Z}_{32}$ is the set of 32-bit integers (or any set of n-bit integers) and *RegNames* is the set of register names reg_0 to reg_{31} (written `$0` to `$31` in machine code). In a state, M and R describe the contents of the *memory* and the *registers* (each is a function from addresses or register names to 32-bit integers), respectively, and tuple $C = (pc, br, dir)$ encapsulates the *control* registers. We do not model input/output facilities. It is assumed that program and data are entered into memory before the machine starts and that the results are read from memory after the machine stops (if it stops). Before loading program and data, all registers and the entire memory are zero-cleared.

Instructions. The design of the abstract machine is closely tied to the design of its reversible instruction set. The instructions fall into the two general classes of *data modification* and *control flow* instructions. As an example of a reversible data modification instruction, consider the addition instruction

```
ADD $3 $4.
```

The effect of executing this instruction is to add to the value in register `$3` the value in register `$4`. This instruction has an inverse interpretation: subtract the value of `$4` from `$3`. A standard instruction usually has the effect of *overwriting* the destination register irreversibly: the original value of the register cannot be recovered from the resulting state. Such irreversible instructions are not allowed in a reversible architecture. A reversible architecture must provide two interpretations of the same instruction depending on the direction bit: the *standard semantics* and the *inverse semantics*.

Depending on *dir*, the current instruction $M(pc)$ is mapped into an instruction implementing its standard or its inverse semantics. This mapping is called *local program inversion* and performed on-the-fly at execution time [9]. It is important that the inversion of an instruction depends only on the local context ('peephole') and, unlike other program inversion methods [10, 13], does not require global analysis of a program. This is key to efficient reversible execution.

As outlined above, control flow instructions interact with the control state in non-trivial ways. The unconditional jump instruction

```
BRA 15
```

has the effect of adding $15 \cdot dir$ to the branch register br. It is important that branch instructions use relative *offsets* for jumps, and not absolute addresses. If $br \neq 0$, the control logic adds $br \cdot dir$ to the program counter pc, instead of irreversibly overwriting the value of pc with an absolute address.

Formal Properties. The main theorems regarding the abstract machine and its instruction set can be summarized as follows (more details can be found in publication [17]). The first theorem states that the semantics of an execution step is forward and backward deterministic if every instruction in the machine's instruction set satisfies certain conditions (every instruction is a reversible update). The second theorem states that an execution step can be reversed in a particular simple way.

Both theorems together guarantee some surprising properties for programs written in reversible machine code. No matter what the program does, regardless of the programmer's intent, execution of the program is guaranteed to be reversible. Furthermore, if the state space is *finite*, programs either terminate or eventually return to the precise starting state, looping forever.

3 Related Work

A great deal of the previous work on reversible machine code suffered from deficiencies. In [4], a history trace was used for reversibility as suggested earlier [12]; in [11] data modification was reversible, but the control logic was unclear and quite probably irreversible. We stress that even if each instruction is reversible, this is *not* sufficient to ensure reversibility of the complete architecture.

In contrast, PISA is the newest and most complete design of a reversible instruction set architecture [15]. The description of the instruction set is informal [7], but our formalization has been shown to be reversible as described[2]. This makes PISA the only truly reversible practical programmable architecture. Reversible logic gates and reversible logic circuits have been studied [8, 6, 5].

A high-level imperative language for reversible programming, Janus, has recently been formalized and confirmed to be reversible [17]. One of the earliest works considering reversible subroutines was [14]. Reversible languages, such as Janus and PISA, allow efficient standard and inverse computation. A general method for inverse computation is the Universal Resolving algorithm [1], which also allows inverse computation of programs that do not implement injective functions, but is less efficient due to the search space.

4 Conclusions

We formalized an abstract machine suitable for reversible computing. We clarified that for machine code to be fully reversible both the underlying control logic as well as each instruction must be reversible, and that forward and backward code can be shared if the machine allows a program to switch between standard (forward) and inverse (backward) computation. We have shown a general

class of instruction sets that are reversible, building on our concept of reversible updates [2] and that PISA, as a member of this class, is reversible.

We posit that reversible computing is sufficiently different from irreversible computing to warrant consideration as a separate paradigm. As such, we suggest a "principles first" approach to reversible machine architectures. Work on a translator from the structured high-level reversible programming language Janus [17] to PISA is currently in progress.

References

1. S. M. Abramov, R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.
2. H. B. Axelsen, R. Glück, T. Yokoyama. Reversible machine code and its abstract processor architecture. In V. Diekert, M. V. Volkov, A. Voronkov (eds.), *Computer Science – Theory and Applications. Proceedings*, LNCS 4649, 56–69. Springer-Verlag, 2007.
3. C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.
4. R. Cezzar. Design of a processor architecture capable of forward and reverse execution. In *Proceeding of IEEE SOUTHEASTCON'91*, Vol. 2, 885–890, 1991.
5. A. De Vos, Y. Van Rentergem, K. De Keyser. The decomposition of an arbitrary reversible logic circuit. *Journal of Physics A: Mathematical and General*, 39(18):5015–5035, 2006.
6. R. P. Feynman. Reversible computation and the thermodynamics of computing (chapter 5). In A. J. G. Hey, R. W. Allen (eds.), *Feynman Lectures on Computation*, 137–184. Addison-Wesley, 1996.
7. M. P. Frank. *Reversibility for Efficient Computing*. PhD thesis, MIT, 1999.
8. E. Fredkin, T. Toffoli. Conservative logic. *Intl. J. Theor. Phy.*, 21:219–253, 1982.
9. R. Glück, M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori (ed.), *Programming Languages and Systems. Proceedings*, LNCS 2895, 246–264. Springer-Verlag, 2003.
10. R. Glück, M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundamenta Informaticae*, 66(4):367–395, 2005.
11. J. S. Hall. A reversible instruction set architecture and algorithms. In *Workshop on Physics and Computation. Proceedings*, 128–134. IEEE Press, 1994.
12. R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.
13. T. Æ. Mogensen. Semi-inversion of guarded equations. In R. Glück, M. Lowry (eds.), *Generative Programming and Component Engineering. Proceedings*, LNCS 3676, 189–204. Springer-Verlag, 2005.
14. E. D. Reilly, F. D. Federighi. On reversible subroutines and computers that run backwards. *Communications of the ACM*, 8(9):557–558, 578, 1965.
15. C. J. Vieri. *Reversible Computer Engineering and Architecture*. PhD thesis, MIT, 1999.
16. C. J. Vieri, M. J. Ammer, M. P. Frank, N. Magoulis, T. Knight. A fully reversible asymptotically zero energy processor. In *Proceedings of the ISCA workshop*, 1998.
17. T. Yokoyama, R. Glück. A reversible programming language and its invertible self-interpreter. In *Partial Evaluation and Program Manipulation. Proceedings*, 144–153. ACM Press, 2007.

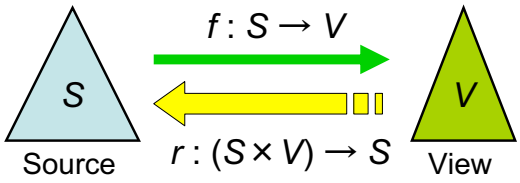
Bidirectionalizing `folds`
(Ongoing Work)

Kazutaka Matsuda*
Zhenjiang Hu*, Keisuke Nakano*
Makoto Hamana**, Masato Takeichi*

*The University of Tokyo
**Gunma University

Bidirectional Transformation

- "Bidirectional Transformation"
  ="View Function" + "Backward Transformation"



[Hegner 90] and so on.

Our Previous Work

- Bidirectionalization Transformation [ICFP 07]
  - Derivation of a backward transformation from a view function
    - Based on Constant Complement Bidirectionalization [Bancilhon & Spyratos 1981]
  - Treeless and Affine language

Restricted Language
• Unable to write "fold" functions

Computation Pattern `fold`

- Widely used computation pattern

```
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

  - Many useful `fold` functions
    - `maximum = foldr max Z`
    - `sort = foldr ins []`
    - `unzip = foldr (¥(a,b) (x,y)->(a:x,b:y)) ([],[])`

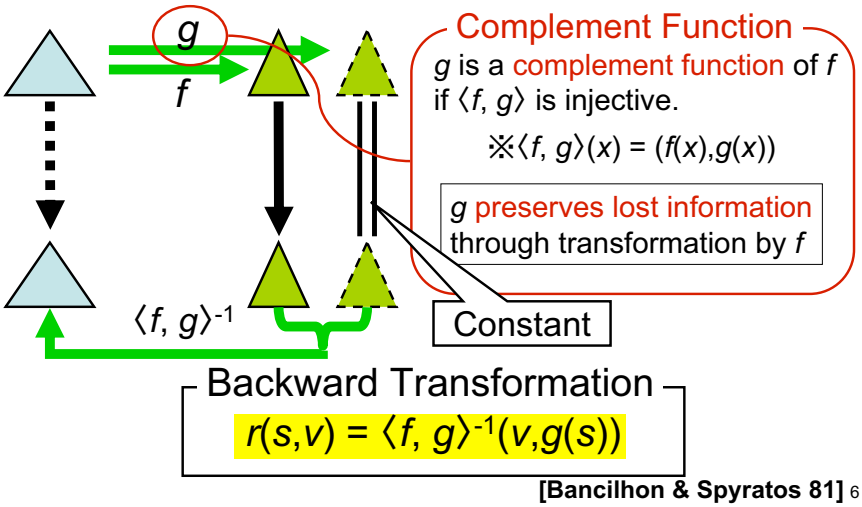
4

Today's Talk

- Bidirectionalization of `folds`
  - in a simple way
- Report of results
  - `maximum`, `sort` and `unzip`

5

Bidirectionalization Based on Constant-Complement



[Bancilhon & Spyratos 81] 6

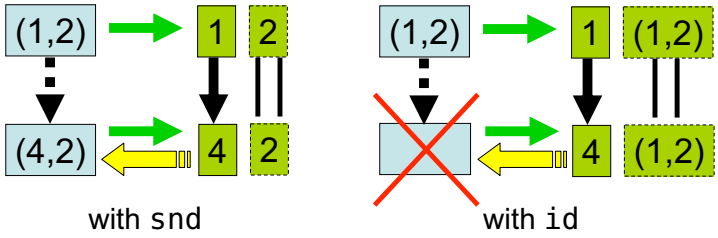
Backward Transformations using Complements

Complements of `fst(Pair(x,y)) = x`

`snd(Pair(x,y)) = y`

`id(Pair(x,y)) = Pair(x,y)`

- A complement yields a backward transformation.

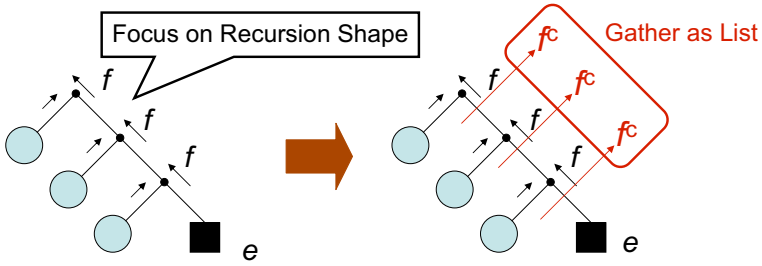


Complement of `foldr`

Given f^c

```
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

```
(foldr f e)ᶜ []     = []
(foldr f e)ᶜ (x:xs) = fᶜ x r:(foldr f e)ᶜ xs
            where r = foldr f e xs
```



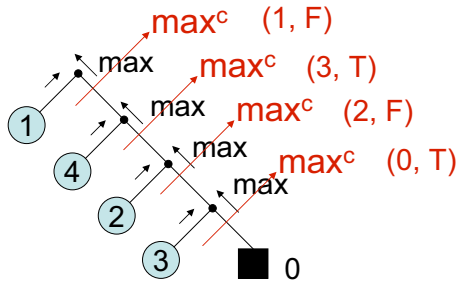
Example: `maximum` (1/3)

```
maximum = foldr max Z

max (S x) (S y) = S (max x y)
max (S x) Z     = S y
max Z y         = x
```

```
maxᶜ (S x) (S y) = (min x y, x > y)
```

Example: `maximum` (2/3)



Note: `max^c x y = (min x y, x > y)`

Observation:
- Remembered all the non-maximum values
- index of maximum ~ history of comparison

Example: `maximum` (3/3)

- Intuitively
  - `maximum^c [1,4,2,3] ~ [1,□,2,3]`

hole for **"maximum"**

Updated value will be inserted here.

Let "r" be the backward transformation with `maximum^c`

```
maximum [1,4,2,3] = 4
r( [1,4,2,3], 5 ) = [1,5,2,3]
r( [1,4,2,3], 2 ) = undefined
```

Updating 4 to 2 is prohibited because the index of the hole changes for [1,2,2,3].

Example: `sort` (1/2)

```
sort = foldr ins []
ins a (b:x) = if a <= b then a:b:x
              else           b:ins a x
ins a []    = [a]
```

```
ins^c a (b:x) = if a <= b then B_LE
                else           B_GT(ins^c a x)
ins^c a []    = B_END
```

Observation:
`ins^c` remembers histories of comparisons
≒ Original Indices

Example: `sort` (2/2)

- Intuitively
  - $\texttt{sort}^c$ `[1,4,2,3]` ~ `[0,2,3,1]`
    Original Indices

Let "r" be the backward transformation with $\texttt{sort}^c$

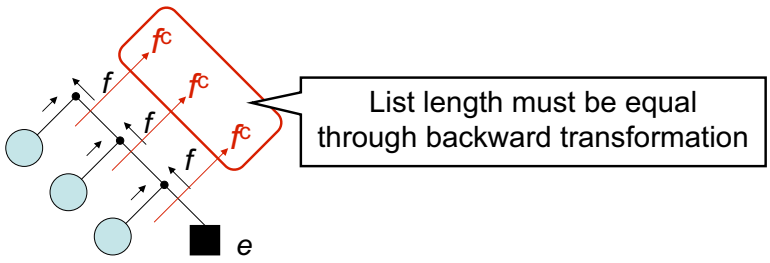
```
sort [1,4,2,3] = [1,2,3,4]
r( [1,4,2,3], [5,6,7,8] )   = [5,8,6,7]
r( [3,4,2,1], [5,6,7,8] )   = [7,8,6,5]
r( [1,4,2,3], [1,2,3] )     = undefined
r( [1,4,2,3], [1,2,3,4,5] ) = undefined
r( [1,4,2,3], [5,8,7,6] )   = undefined
```

13

Property of Derived Complements

- Insertions and deletions are prohibited.

```
(foldr f e)^c []     = []
(foldr f e)^c (x:xs) = f^c x r:(foldr f e)^c
            where r = foldr f e xs
```



14

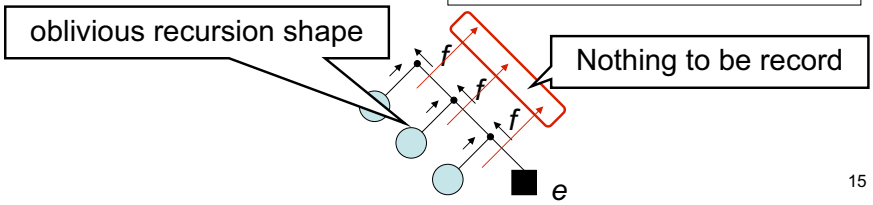
Sufficient Condition for Insertions and Deletions

- Sufficient condition for insertion and deletions
  ⇐ Sufficient condition for injectivity of `foldr`
  - "f" is injective
  - Range of "f" and "e" do not overlap

Constant Function

```
(foldr f e)^c xs = C
```

Computable
if "f" and "e" are written in
the language in our previous work
[ICFP 07]



15

Example: unzip

- unzip xs =
  foldr (¥(a,b) (x,y)->(a:x,b:y))
  ([],[]) xs
  - The sufficient condition for injectivity holds.
    - Complement is a constant function.

Let "r" be the backward transformation with $\texttt{unzip}^c$

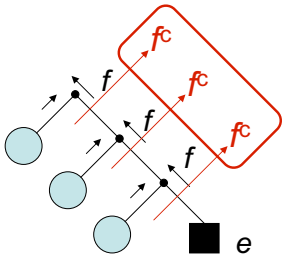
```
s = [(1,2), (3,4)]
unzip s = ([1,3],[2,4])
r(s,([9,7],[8,6])) = [(9,8),(7,6)]
r(s,([1],[2,4]))   = undefined
```

([1],[2,4]) is not in the range of unzip.

16

Summary for foldr

- Bidirectionalization of foldrs
  - in a simple way
  - Presented results
    - maximum, sort and unzip



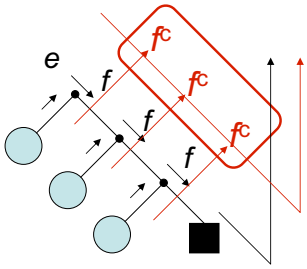
17

Computation Pattern foldl

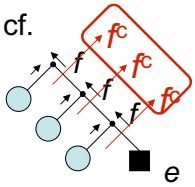
Given f^c

```
foldl f e []     = e
foldl f e (x:xs) = foldl f (f x e) xs
```

```
(foldl f)^c (e,e') []     = e'
(foldl f)^c (e,e') (x:xs) = (foldl f)^c (s,t) xs
            where (s,t) = (f x e, f^c x e:e')
```



cf.



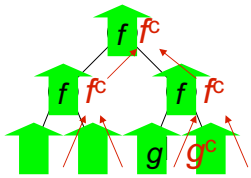
18

Computation Pattern $\texttt{fold}_{BIN}$

Given f^c, g^c

```
fold_BIN f g (L x)     = g x
fold_BIN f g (N n l r) =
         f n (fold_BIN f g l) (fold_BIN f g r)
```

```
(fold_BIN f g)^c (L x)     = g^c x
(fold_BIN f g)^c (N n l r) = N (f^c n s t) fl fr
       where s  = fold_BIN f g l
             t  = fold_BIN f g r
             fl = (fold_BIN f g)^c l
             fr = (fold_BIN f g)^c r
```



19

Conclusion & Future Work

- Conclusion
  - Bidirectionalization of `folds`
    - in a simple way
    - also `foldl` and more generic `folds`
- Future work
  - Formalization
    - Categories
  - Checking the effectiveness of this strategy

20

Relational Reasoning for Recursive Types and References

Nina Bohr
joint work with Lars Birkedal

IT University of Copenhagen (ITU)

Overview

Proof method for contextual equivalence of expressions

in a λ-calculus with recursive types and generel references.

Uses a parameterized logical relation

over a denotational semantics.

Supports local reasoning.

Extends and builds on work of:
Nick Benton and Benjamin Leperchey, Andrew Pitts, Ian Strak, Mark Shinwell.

Technical development

1. Language definiton.
   Operational semantics given by termination judgements.
2. Denotational semantics in a recursive domain.
3. Definition of local parameters expressing properties of finite parts of stores.
   Definition of parameters as finite sets of local parameters together with a finite visible area of stores.
4. Definition of a parameterized logical relation ∇ over the recursive domain.
5. Definition of a binary relation between denotations of open terms. Relatedness in ∇^{Γ} under a "minimal" parameter implies contextual equivalence.

Language. Types and Terms

Types

Value types $\tau ::= \alpha \mid unit \mid int \mid \tau \times \tau \mid \tau + \tau \mid \tau\ ref \mid \tau \to T\tau \mid \mu\alpha.\tau$

Computation types $T\tau$.

Terms

Values $V ::= x \mid \underline{n} \mid \underline{l} \mid () \mid (V, V') \mid in_i V \mid rec\ f(x:\tau) = M \mid fold\ V$

Computations $M ::= VV' \mid let\ x \Leftarrow M\ in\ M' \mid val\ V \mid \pi_i V \mid$
$ref\ V \mid !V \mid V := V' \mid case\ V\ of\ in_1 x_1 \Rightarrow M_1;\ in_2 x_2 \Rightarrow M_2 \mid V = V'$
$V + V' \mid iszero\ V \mid unfold\ V$

store $\Sigma \in (\mathbb{L} \rightharpoonup_{fin} closed\ Values)$,

store type $\Delta \in (\mathbb{L} \rightharpoonup_{fin} closed\ Value\ types)$

Operational semantics

Continuation terms in x

$$\frac{}{val\ x \in Cont_x} \qquad \frac{fv(M) \subseteq \{x\} \qquad K \in Cont_y}{\text{let } y \Leftarrow M \text{ in } K \in Cont_x}$$

Termination judgements:

$$\Sigma,\ let\ x \Leftarrow M\ in\ K \downarrow$$

Σ is a store
M is a closed computation
K is a continuation.

Contextual equivalence

Typed Contexts

$$C[.] : (\Delta; \Gamma \vdash \gamma) \Rightarrow (\Delta; - \vdash T\tau)$$

means that whenever $\Delta; \Gamma \vdash G : \gamma$ then $\Delta; - \vdash C[G] : T\tau$.

Definition: Contextual equivalence

If $\Delta; \Gamma \vdash G_1 : \gamma$ and $\Delta; \Gamma \vdash G_2 : \gamma$ then

$\Delta; \Gamma \vdash G_1 =_{ctx} G_2$ means

$\forall \tau. \forall C[.] : (\Delta; \Gamma \vdash \gamma) \Rightarrow (\Delta; - \vdash T\tau). \forall \Sigma : \Delta.$
$\Sigma, let\ x \Leftarrow C[G_1]\ in\ val\ x \downarrow \iff \Sigma, let\ x \Leftarrow C[G_2]\ in\ val\ x \downarrow$

Denotational Semantics

There exists a minimal invariant recursive domain

$$\mathbb{D} \;=\; (\mathbb{V}, \mathbb{K}, \mathbb{M}, \mathbb{S}) \in \; (\mathrm{FM} - \mathrm{Cpo}_{\perp})^4$$

and isomorphism $i : F(\mathbb{D}, \mathbb{D}) \cong \mathbb{D}$ as

$$\begin{array}{lcl} \mathbb{V} & \cong & \mathbb{1}_{\perp} \oplus \mathbb{Z}_{\perp} \oplus \mathbb{I}_{\lightning\perp} \oplus (\mathbb{V} \oplus \mathbb{V}) \oplus (\mathbb{V} \otimes \mathbb{V}) \oplus (\mathbb{V} \multimap \mathbb{M})_{\perp} \oplus \mathbb{V} \\ \mathbb{K} & \cong & (\mathbb{S} \multimap (\mathbb{V} \multimap \{\top\}_{\perp})) \\ \mathbb{M} & \cong & (\mathbb{K} \multimap (\mathbb{S} \multimap \{\top\}_{\perp})) \\ \mathbb{S} & \cong & (\mathbb{I}_{\lightning\perp} \multimap \mathbb{V}) \end{array}$$

Existence of $\mathbb{D}$:
proof adapted from Andrew Pitts and Mark Shinwell.

Category FM-Cpo$_{\perp}$. Atom set $\mathbb{I}_{\lightning}$. Objects: pointed FM-cpos.
Morphisms: strict empty-supported FM-continous functions.
Function Space: strict finitely supported FM-continous functions.

Soundness and Adequacy

Soundness:

$$(\,\Sigma, let\ x \Leftarrow M\ in\ K\ \downarrow\,) \quad \Longrightarrow \quad (\,[\![\Delta;\vdash M : T\tau]\!]\{\}[\![\Delta;\vdash K : (x:\tau)^{\top}]\!]S = \top\,)$$

Proof by induction on termination judgement.

Adequacy:

$$(\,[\![\Delta;\vdash M : T\tau]\!]\{\}[\![\Delta;\vdash K : (x:\tau)^{\top}]\!]S = \top\,) \quad \Longrightarrow \quad (\,\Sigma, let\ x \Leftarrow M\ in\ K\ \downarrow\,)$$

Proof by a logical relation between denotational and operational semantics.

Existence of the relation is proved by methods adapted from Pitts/Shinwell.

Proving more contextual equivalences

Aim:

Definition of a parameterized logical relation
between denotations of terms

which implies contextual equivalence
in the operational semantics

and supports modular reasoning

Definitions preparing for Local Reasoning

An accessibility map A is a function from states to finite sets of locations.

$$A : \mathbb{S} \to \mathcal{P}_{fin}(\mathbb{L})$$

such that, if two states S_1, S_2 are identical on $A(S_1)$, then $A(S_1) = A(S_2)$.

A simple state relation P can only "take into consideration"
some finite parts of the states, given by a pair of accessibility maps.

$$P = (\hat{p}, A_{p1}, A_{p2})$$

where $\hat{p}$ is a binary relation of states.

Local Parameter

A a local parameter will be used to express a hidden invariant of stores.
A local parameter "owns" a finite set of locations in each side.

- Ordinary local vm-parameter $r = q = \left(\langle P_{11}, LL_{11}\rangle \vee \ldots \vee \langle P_{1m_1}, LL_{1m_1}\rangle\right)$

  P_{ij} is a simple state relation. LL_{ij} is a finite set of location pairs and closed value types.

Local vm-parameter $r = q_1 \,\overline{\wedge}\, \ldots \,\overline{\wedge}\, q_k =$
$\left(\langle P_{11}, LL_{11}\rangle \vee \ldots \vee \langle P_{1m_1}, LL_{1m_1}\rangle\right) \overline{\wedge} \ldots\ldots \overline{\wedge} \left(\langle P_{k1}, LL_{k1}\rangle \vee \ldots \vee \langle P_{km_k}, LL_{km_k}\rangle\right)$

- Local k-parameter $(rk) = (r|q_i)$ is a local vm-parameter together with a choiche of one q_i from r.
- Local s-parameter $(rks) = (r|q_i|\langle P_{ij}, LL_{ij}\rangle)$ is a local k-parameter together with a choiche of one P_{ij} from q_i.

Parameters

Δ is a store type $\quad r = \{r_1, .., r_n\}$ is a finite set of local parameters.

vm-parameter $\Delta r = \Delta\{r_1, .., r_n\}$

k-parameter $\Delta(rk) = \Delta\{(r_1|q_1), .., (r_n|q_n)\}$

s-parameter $\Delta(rks) = \{(r_1|q_1|\langle P_1, LL_1\rangle), .., (r_n|q_n|\langle P_n, LL_n\rangle)\}$

Orders on parameters

- $\Delta' r' \rhd \Delta r \stackrel{def}{\Longleftrightarrow}$
  $\Delta' \supseteq \Delta$ and $r' = r \uplus \{r_{n+1} \ldots r_{n+m}\}$ and $r_{n+1} \ldots r_{n+m}$ are ordinary.

– $\Delta'(rk') \rhd \Delta(rk) \stackrel{def}{\Longleftrightarrow} (rk') \supseteq (rk)$ and $\Delta' r' \rhd \Delta r$.

- $\Delta' r' \blacktriangleright \Delta r \stackrel{def}{\Longleftrightarrow} \Delta' \supseteq \Delta$ and
  $r' = \{r'_1 \ldots r'_n, r'_{n+1} \ldots r'_{n+m}\}$ and $r = \{r_1, .., r_n\}$ and $\forall i \in \{1..n\}. r'_i \succeq r_i$.

  where

  Order $\succeq$ on local parameters is defined by removal of $q's$:
  $(q_1 \mathbin{\overline{\wedge}} \ldots \mathbin{\overline{\wedge}} q_k) \;\succeq\; (q_1 \mathbin{\overline{\wedge}} \ldots \mathbin{\overline{\wedge}} q_k \mathbin{\overline{\wedge}} \ldots \mathbin{\overline{\wedge}} q_{k+m})$

Invariant relation ∇

Theorem :

There exists a relational lifting of the domain constructing functor F
to $\mathcal{R}(\mathbb{D})^{op} \times \mathcal{R}(\mathbb{D}) \to \mathcal{R}(F(\mathbb{D}, \mathbb{D}))$
and
an admissible relation $\nabla = (\nabla_V, \nabla_K, \nabla_M, \nabla_S) \in \mathcal{R}_{adm}(\mathbb{D})$
satisfying the equations on the following slides
and $(i, i) : F(\nabla, \nabla) \subset \nabla \;\wedge\; (i^{-1}, i^{-1}) : \nabla \subset F(\nabla, \nabla)$.

Invariant relation ∇. Computations

$$
\begin{array}{lll}
\nabla_M & = & \{(m'_1,\ m_1,\ m'_2,\ m_2,\ T\tau,\ \Delta r) \mid \\
 & & \quad m'_1 \sqsubseteq m_1 \ \wedge\ m'_2 \sqsubseteq m_2 \ \wedge \\
 & & \quad \forall \Delta' r' \blacktriangleright \Delta r.\ \forall (rk') \in (r')^K.\ \forall (rks') \in (rk')^S. \\
 & & \quad \forall (k'_1,\ k_1,\ k'_2,\ k_2,\ (x:\tau)^\top,\ \Delta'(rk')) \in \nabla_K. \\
 & & \quad \forall (S'_1,\ S_1,\ S'_2,\ S_2,\ \Delta'(rks')) \in \nabla_S\ . \\
 & & \qquad (m'_1 k'_1 S'_1 = \top \Rightarrow m_2 k_2 S_2 = \top) \wedge \\
 & & \qquad (m'_2 k'_2 S'_2 = \top \Rightarrow m_1 k_1 S_1 = \top)\ \}
\end{array}
$$

$$
\begin{array}{lll}
\nabla_K & = & \{(k'_1,\ k_1,\ k'_2,\ k_2,\ (x:\tau)^\top,\ \Delta(rk)) \mid \\
 & & \quad k'_1 \sqsubseteq k_1 \ \wedge\ k'_2 \sqsubseteq k_2 \ \wedge \\
 & & \quad \forall \Delta'(rk') \rhd \Delta(rk).\ \forall (rks') \in (rk')^S. \\
 & & \quad \forall (S'_1,\ S_1,\ S'_2,\ S_2,\ \Delta'(rks')) \in \nabla_S. \\
 & & \quad \forall (v'_1,\ v_1,\ v'_2,\ v_2,\ \tau,\ \Delta' r') \in \nabla_V. \\
 & & \qquad (k'_1 S'_1 v'_1 = \top \Rightarrow k_2 S_2 v_2 = \top) \wedge \\
 & & \qquad (k'_2 S'_2 v'_2 = \top \Rightarrow k_1 S_1 v_1 = \top)\ \}
\end{array}
$$

The invariant relation ∇. (Values)

$$\nabla_V = \{(\bot,\ v_1,\ \bot,\ v_2,\ \tau,\ \Delta r)\ \} \cup$$

$$\{(v_1',\ v_1,\ v_2',\ v_2, \tau, \Delta r) \mid v_1' \sqsubseteq v_1 \neq \bot\ \wedge\ v_2' \sqsubseteq v_2 \neq \bot\ \} \cap$$

$$(\ \{(v_1',\ n,\ v_2',\ n,\ int,\ \Delta r)\ \} \cup$$

$$\ldots\ldots$$

$$\{(f_1',\ f_1,\ f_2',\ f_2,\ \tau \to \mathsf{T}\tau',\ \Delta r) \mid$$
$$\forall \Delta' r' \blacktriangleright \Delta r,\ (v_1',v_1,\ v_2',\ v_2,\ \tau,\ \Delta' r') \in \nabla_V.$$
$$(f_1' v_1',\ f_1 v_1,\ f_2' v_2',\ f_2 v_2,\ \mathsf{T}\tau',\ \Delta' r') \in \nabla_M\ \}\ )$$

The invariant relation ∇. (States)

$$\nabla_S\ =\ \{(\bot,\ S_1,\ \bot,\ S_2,\ \Delta(rks))\ \} \cup$$
$$\{(S_1',\ S_1,\ S_2',\ S_2,\ \Delta(rks)) \mid (rks) = \{(\mathfrak{r}_1|q_1|(P_1, LL_1)), \ldots, (\mathfrak{r}_n|q_n|(P_n, LL_n)\} \wedge$$

$$S_1' \sqsubseteq S_1 \neq \bot\ \ \wedge\ S_2' \sqsubseteq S_2 \neq \bot \wedge$$

$$\forall l \in dom(\Delta).(S_1' l,\ S_1 l,\ S_2' l,\ S_2 l,\ \Delta l,\ \Delta r) \in \nabla_V \wedge$$

$$dom(\Delta) \cap A_{r1}(S_1) = \emptyset\ \wedge\ dom(\Delta) \cap A_{r2}(S_2) = \emptyset \wedge$$

$$\forall i \neq j \in \{1..n\}.\ A_{\mathfrak{r}i1}(S_1) \cap A_{\mathfrak{r}j1}(S_1) = \emptyset\ \wedge\ A_{\mathfrak{r}i2}(S_2) \cap A_{\mathfrak{r}j2}(S_2) = \emptyset \wedge$$

$$\forall (P, LL) \in (pks).$$
$$(S_1, S_2) \in P\ \wedge\ \forall (l_1, l_2, \tau) \in LL.(S_1' l_1, S_1 l_1, S_2' l_2, S_2 l_2, \tau, \Delta r) \in \nabla_V\ \}$$

Relating denotations of open expressions

Binary relation ∇^Γ between denotations of terms.

Used for proofs of contextual equivalence.

For ordinary parameter Δr:

- $(v_1, v_2, \tau, \Delta r) \in \nabla_V^\Gamma \stackrel{def}{\iff}$
  $\forall \Delta' r' \blacktriangleright \Delta r. \forall (v_{1i}', v_{1i}, v_{2i}', v_{2i}, \tau_i, \Delta' r') \in \nabla_V, i = 1, .., n..$
  $(v_1(\overline{v_{1i}'}), v_1(\overline{v_{1i}}), v_2(\overline{v_{2i}'}), v_2(\overline{v_{2i}}), \tau, \Delta' r') \in \nabla_V.$

- $(m_1, m_2, T\tau, \Delta r) \in \nabla_M^\Gamma \stackrel{def}{\iff}$
  $\forall \Delta' r' \blacktriangleright \Delta r. \forall (v_{1i}', v_{1i}, v_{2i}', v_{2i}, \tau_i, \Delta' r') \in \nabla_V, i = 1, .., n.$
  $(m_1(\overline{v_{1i}'}), m_1(\overline{v_{1i}}), m_2(\overline{v_{2i}'}), m_2(\overline{v_{2i}}), T\tau, \Delta' r') \in \nabla_M.$

If $(m_1, m_2, T\tau, \Delta r) \in \nabla_M^\emptyset$ and
$(k_1, k_1, k_2, k_2, (x : \tau)^\top, \Delta r) \in \nabla_K$ and $(S_1, S_1, S_2, S_2, \Delta r) \in \nabla_S$
then $m_1 k_1 S_1 = \top \iff m_2 k_2 S_2 = \top$

Fundamental Theorem

$Theorem$:

- Typing rules preserve the ∇^{Γ} relation.
- For all ordinary parameters Δr it holds that
  - $([\![\Delta;\Gamma \vdash V : \tau]\!], [\![\Delta;\Gamma \vdash V : \tau]\!], \tau, \Delta r) \in \nabla^{\Gamma}_{V}$,
  - $([\![\Delta;\Gamma \vdash M : T\tau]\!], [\![\Delta;\Gamma \vdash M : T\tau]\!], T\tau, \Delta r) \in \nabla^{\Gamma}_{M}$.

Contextual Equivalence

$Theorem$:

Let $C[\cdot] : (\Delta;\Gamma \vdash \gamma) \Rightarrow (\Delta;\vdash T\tau')$ be a context, ($\gamma = \tau$ or $\gamma = T\tau$)

If $([\![\Delta;\Gamma \vdash G_1 : \gamma]\!], [\![\Delta;\Gamma \vdash G_2 : \gamma]\!], \gamma, \Delta id_{\emptyset}) \in \nabla^{\Gamma}_{X}$, $(_{X=_{V,M}})$

then $\forall \Sigma : \Delta.\ (\Sigma, let\ x \Leftarrow C[G_1]\ in\ val\ x \downarrow \Longleftrightarrow \Sigma, let\ x \Leftarrow C[G_2]\ in\ val\ x \downarrow)$.

Conclusion

- *A local relational proof method for establishing contextual equivalence of expressions in a language with recursive types and general references, extending earlier work of Benton and Leperchey.*
- *The proof of existence of the logical relation is quite intricate because of the interplay between recursive types and local parameters for reasoning about higher-order store.*
- *The method is easy to use on examples: the only non-trivial steps are to guess the right local parameters — but since the local parameters express the intuitive reason for contextual equivalence, the non-trivial steps are really fairly straightforward.*

References

Selected references.

- *Nina Bohr and Lars Birkedal: Relational Reasoning for Recursive Types and References. APLAS 06 Lecture Notes in Computer Science 4279*
- *Nick Benton and Benjamin Leperchey: Relational Reasoning in a Nominal Semantics for Storage. TLCA'05 Lecture Notes in Computer Science 3461*
- *Andrew Pitts: Relational Properties of Domains. Information and Computation 127, 1996,*
- *Mark Shinwell: The Fresh Approach: Functional Programming with Names and Binders. Computer Laboratory, Cambridge University, Phd Thesis 2004*

Associativity for Parallel Tree Computation

Kiminori Matsuzaki

University of Tokyo

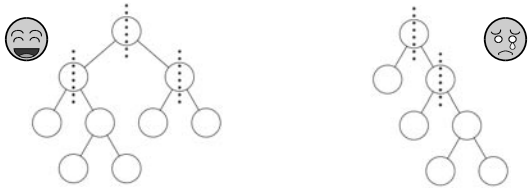
Target Computation

- Input: a huge binary tree
  - Its shape is fixed before computation
    - ✓ Non-example: dynamically generated trees (e.g. game trees)
- Output: a value or another tree
  - Examples:
    - ✓ Computing height of binary tree
    - ✓ Several optimization problems on trees
    - ✓ Queries on tree-structured data

2

(Naive) Divide and Conquer

- Divide a tree into two subtrees at the root



☺ Obvious parallelism

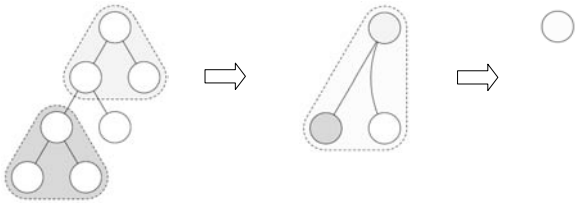
☺ Easy to develop parallel programs

☹ Inefficient if trees are ill-balanced

3

Parallel Tree Contraction [Miller and Reif 85]

- Apply local contractions in parallel
  - Shunt contraction algorithm [Abrahamson et al. 89]



• Algorithms are defined by procedures

• No formalization based on data structure

4

This Talk

- A formalization of parallel tree computation from the viewpoint of data structures
  - Flexible division of binary trees
    → Ternary-tree representation
  - "Associativity" on ternary-tree representation

Idea

Analogy to parallel list computation

  - Several results & Open problems

5

Formalization of Parallel List Computation

Two definitions of Lists

- Cons list

  `data CList a = [ ] | a : (CList a)`

  - Element is added/consumed to/from the head

- Join list

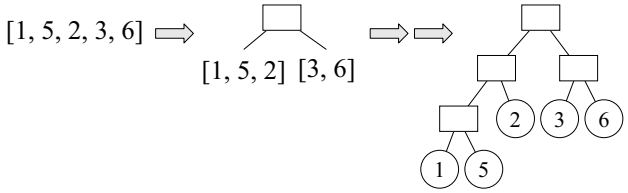
  `data JList a = [a] | (JList a) ++ (JList a)`

  - A list can be divided at any point
    ← Associativity of ++
    - ✓ E.g. [1, 5, 2, 3, 6] = [1] ++ [5, 2, 3, 6]
      = [1, 5, 2] ++ [3, 6]

7

Binary-Tree Representation

- Recursive division of a join list
  → Binary-tree representation



- Divide-and-conquer on binary-tree representation
  → A parallel algorithm on join lists
  - Associativity is required

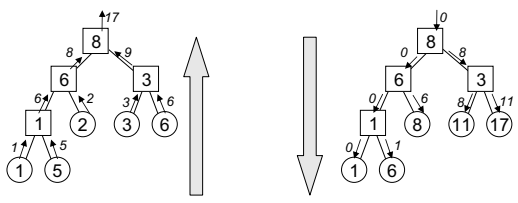
8

Parallel Prefix-Sums [Stone 73]

- Prefix-sums (scan)

  `scan (+) [1, 5, 2, 3, 6] = [1, 6, 8, 11, 17]`

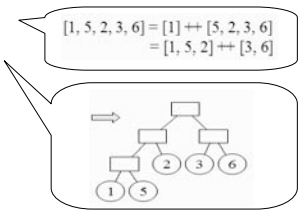
- Two-pass algorithm on binary-tree representation



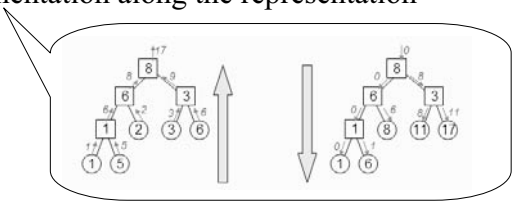
9

Three Key Points in Parallelism

- Flexible division and its representation



- Associativity
  - For correctness
- Implementation along the representation

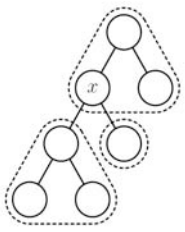


10

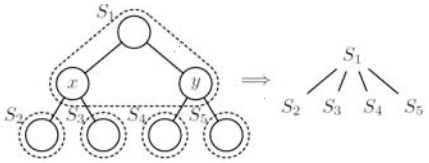
Formalization of Parallel Tree Computation

Flexible Division of Binary Tree

- Divide at any internal node
  - Terminal node (x)
  - Segment (dashed line)
- The only condition:
  - A segment has at most one terminal node



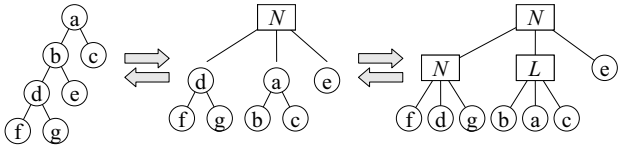
→ To keep global structure binary tree



12

Ternary-Tree Representation

- Represent recursive division of a tree
  - Single division → Thee segments

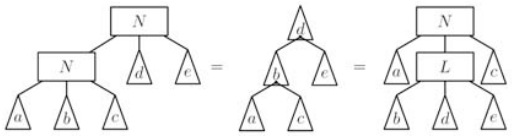


  - Introduce labels to restore the original tree
    - ✓ *N*: No terminal node exists
    - ✓ *L*: Terminal node exists in the left subtree.
    - ✓ *R*: Terminal node exists in the right subtree

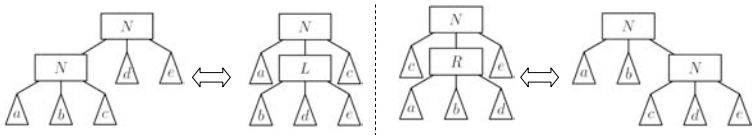
13

Multiple Representations for Single Tree

- Different order of divisions yields different ternary-tree representation



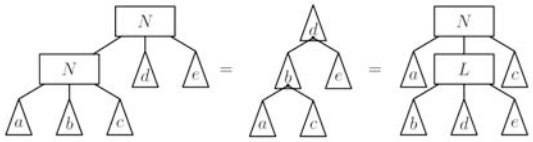
- 6 equations for two successive divisions
  - In fact, the following 2 equations are essential!



14

Tree Associativity

- The 2 equations define the equivalence of two ternary-tree representations



- Generalize labels into three functions

Definition (Tree Associativity): Functions g_n, g_l, g_r are said tree associative if the following two equations hold.

$$g_n(g_n(a, b, c), d, e) = g_n(a, g_l(b, d, e), c)$$
$$g_n(a, b, g_n(c, d, e)) = g_n(c, g_r(a, b, d), e)$$

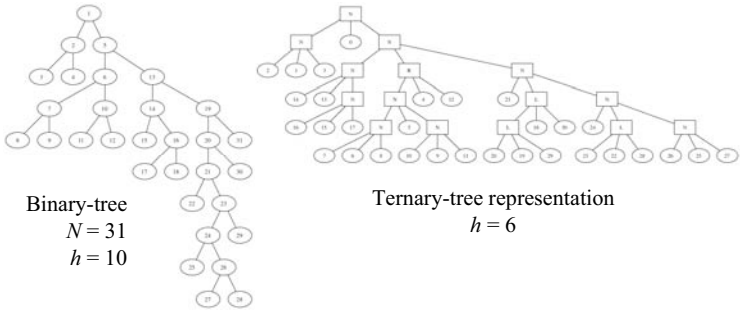
15

Obtained Results

Balanced Ternary-Tree Representation

Binary tree of N nodes

➔ Ternary tree of height h = 1.71 log (N+1) – 1.42



Binary-tree
N = 31
h = 10

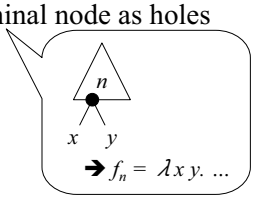
Ternary-tree representation
h = 6

17

Existence of Tree Associative Functions

- Tree associative functions always exist unless we mind the efficiency
  - Use functions (closures) for local results
  - Assume two children of the terminal node as holes

$g_n\,(l, f_n, r) = f_n\,(l, r)$
$g_l\,(f_l, f_n, r) = \lambda x\,y.\, f_n\,(f_l\,(x, y), r)$
$g_r\,(l, f_n, f_r) = \lambda x\,y.\, f_n\,(l, f_r\,(x, y))$



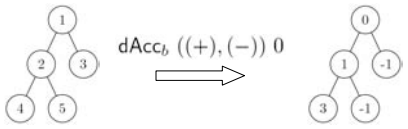
- Condition of tree contraction [Abrahamson et al. 89] = Tree associative functions in a fixed form

18

Tree Accumulations [Skillicorn 96]

- Upwards Accumulation

- Downwards Accumulation



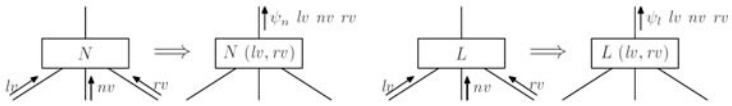
If there exist tree associative functions,
we can implement overall computation in parallel.

19

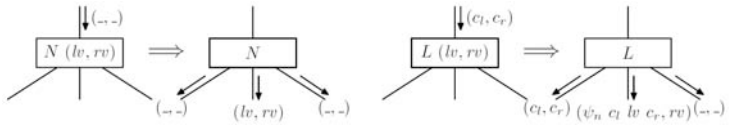
Upwards Accumulation on Ternary Tree

(Shown for internal nodes *N* and *L* only)

- Bottom-up sweep



- Top-down sweep



If there exist tree associative functions,
we can implement overall computation in parallel

20

Open Problems

Framework of "Parallel Data Structures"

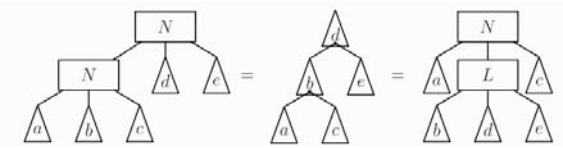
- How can we formalize a unified framework?

List	Binary-tree representation + Associativity
Matrix	Binary-tree representation + Abide property [Bird 96], [Emoto et al. 06]
Binary Tree	Ternary-tree representation + Tree associativity
General Tree	??

Abide Property: $\begin{matrix} A \oplus B \\ \ominus \\ C \oplus D \end{matrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{matrix} A & \\ \ominus & \oplus \\ C & \end{matrix} \begin{matrix} B \\ \ominus \\ D \end{matrix}$

Dynamic Balancing Algorithm

- Equations can be considered "Rotations"



- Dynamic balancing like AVL tree/red-black tree
  ➔A new scheduling for parallel tree computation

Related Work

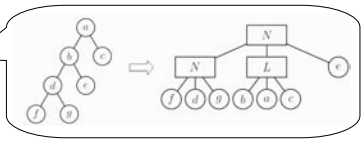
- Decomposition tree [Fujiwara et al. 00]
  - Dividing a binary tree at edges
  - Insufficient information for restoring original tree
    → Limited applications

- Parallelism on other data structures
  - Lists: Binary-tree representation
    + associativity
  - Matrices: Binary-tree representation
    + abide property [Bird 96], [Emoto et al. 06]

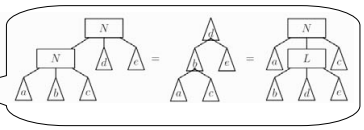
Conclusion

A Formalization of Parallel Tree Computation

- Flexible division of binary trees
  - ➔ Ternary-tree representation



- Tree associativity



- Several known results and open problems

25

Logic-Based Modelling and Analysis of Embedded Systems

Gourinath Banda
John Gallagher
Programming, Logic and Intelligent Systems Group
University of Roskilde, Denmark

Supported by EU framework 5 project ASAP (2002-2006) and Danish Research Council project SAFT (from 2007)

Computer Science, building 42.2
Roskilde University
Universitetsvej 1
P.O. Box 260
DK-4000 Roskilde
Denmark
Phone: +45 4674 2000
Fax: +45 4674 3072
www.dat.ruc.dk

From Embedded System Design to CLP

- Constraint logic programs (CLP) used as a modelling language
  - building on work by Jaffar *et al.* Delzano & Podelski, Gupta *et al.,* Flanagan, Leuschel, Ramakrishnan *et al., ...*
- discrete/continuous state variables
- deterministic/non-deterministic
- finite/infinite state space
- finite/infinite traces
- structure/functionality of systems
- reasoning backwards/forwards
- safety/liveness

CLP transformation and analysis

- CLP transformations are applied to bring the model to the form of a transition system
  - from component-based to transition system form
  - from generic schema to application-specific program
  - transformations to modify the transition system (e.g. from forwards to backwards)
  - transformations to instrument the transition system, make dependencies explicit, show traces,...
- CLP analysis tools applied
  - primarily *abstract interpretation* of the declarative semantics

CLP declarative semantics

- We consider primarily the declarative semantics of CLP programs

- The least model of a program P
  - can be obtained as the least fixed point of an immediate consequences operator T_P
  - given a set of atomic formulae I,
    $T_P(I)$ is the set of all atomic formulae derivable in P by one bottom-up inference step.
  - $M[P] = lfp(T_P)$

Transition Systems as CLP programs

```
state(S2) :-
        transition(S1,S2),
        state(S1).
state(S0) :-
        init(S0).
```

Given a set of transitions, least model contains the reachable states.

```
qstate(S1) :-
        transition(S1,S2),
        qstate(S2).
qstate(S) :-
        queryState(S).
```

An inverted program (query-answer program, "magic set" program)

Least model contains states that "lead to" a given query state.

Representing traces

```
state([S2,S1|Trace]) :-
        transition(S1,S2),
        state([S1|Trace]).
state([S0]) :-
        init(S0).
```

least model contains the finite traces (sequences of states) starting from an initial state.

```
qstate([S1,S2|Trace]) :-
        transition(S1,S2),
        qstate([S2|Trace]).
qstate([S]) :-
        queryState(S).
```

least model contains finite traces that "end with" a given query state.

Representing dependencies

```
state(S2,S0) :-
        transition(S1,S2),
        state(S1,S0).
state(S0,S0) :-
        init(S0).
```

Least model contains the pairs of states $<S_k,S_0>$ such that S_k is reachable from S_0.

Dependencies of a state on the initial state variables can be analysed.

```
qstate(S1,S) :-
        transition(S1,S2),
        qstate(S2,S).
qstate(S,S) :-
        queryState(S,S).
```

Least model contains the pairs of states $<S_m,S_n>$ such that a given state S_n is reached from S_m.

Pre-conditions on a given state can be analysed.

Program specialisation

- The above schemata are specialised w.r.t. given transition relations.
- Polyvariant specialisation can also yield a different state predicate for each location.

- We use Leuschel's Logen off-line specialisation tool.

Motivating Example

```
(0) while (i < n) {
(1)    i++; (2)
     } (3)
```

```
init([0,I,N]) :-
  I=0,
  N >=0.

transition([0,I,N],[1,I,N]) :-
          I < N.
transition([2,I,N],[1,I,N]) :-
          I < N.
transition([1,I,N],[2,I1,N]) :-
          I1 = I+1.
transition([0,I,N],[3,I,N]) :-
          I >= N.
transition([2,I,N],[3,I,N]) :-
          I >= N.
```

Specialisations

```
state__1__1(A,B) :-
        A=0,
        B>=0.

state__1__2(A,B) :-
        A<B,
        state__1__1(A,B).
state__1__2(A,B) :-
        A<B,
        state__1__3(A,B).

state__1__3(A,B) :-
        A=C+1,
        state__1__2(C,B).

state__1__4(A,B) :-
        A>=B,
        state__1__1(A,B).
state__1__4(A,B) :-
        A>=B,
        state__1__3(A,B).
```

forwards

```
qstate__2__5(A,B) :-
        A<B,
        qstate__2__6(A,B).
qstate__2__5(A,B) :-
        A>=B,
        qstate__2__8(A,B).

qstate__2__6(A,B) :-
        C=A+1,
        qstate__2__7(C,B).

qstate__2__7(A,B) :-
        A<B,
        qstate__2__6(A,B).
qstate__2__7(A,B) :-
        A>=B,
        qstate__2__8(A,B).

qstate__2__8(A,B) :-
        A<B.
```

backwards

```
state2__3__9(A,B,0,A,B) :-
        A=0,
        B>=0.

state2__3__10(A,B,C,D,E) :-
        A<B,
        state2__3__9(A,B,C,D,E).
state2__3__10(A,B,C,D,E) :-
        A<B,
        state2__3__11(A,B,C,D,E).

state2__3__11(A,B,C,D,E) :-
        A=F+1,
        state2__3__10(F,B,C,D,E).

state2__3__12(A,B,C,D,E) :-
        A>=B,
        state2__3__9(A,B,C,D,E).
state2__3__12(A,B,C,D,E) :-
        A>=B,
        state2__3__11(A,B,C,D,E).
```

forwards dependencies

```
qstate2__4__13(A,B,C,D,E) :-
        A<B,
        qstate2__4__14(A,B,C,D,E).
qstate2__4__13(A,B,C,D,E) :-
        A>=B,
        qstate2__4__16(A,B,C,D,E).

qstate2__4__14(A,B,C,D,E) :-
        F=A+1,
        qstate2__4__15(F,B,C,D,E).

qstate2__4__15(A,B,C,D,E) :-
        A<B,
        qstate2__4__14(A,B,C,D,E).
qstate2__4__15(A,B,C,D,E) :-
        A>=B,
        qstate2__4__16(A,B,C,D,E).


qstate2__4__16(A,B,3,A,B) :-
        A<B.
```

backwards dependencies

Analysis with a convex polyhedral tool

- Approximation of states at (3) (initial state n >= 0).

  state__1__4(A,B) :- [1*B>=0,-1*A+1*B> -1,1*A+ -1*B>=0]

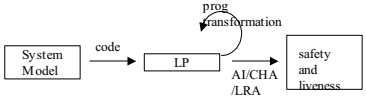
- Approximation of states at (1) (reaching point (3) with i<n.

  qstate__2__5(A,B) :- [empty]

  I.e. the given state cannot be reached from point (0)

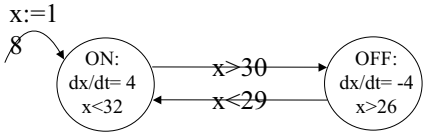
Embedded Systems Specification

- Quest to develop correct safety critical systems resulted in
  - **Formal specification languages**: Timed Automata, Linear Hybrid Automata
  - **Generative programming:** SIGNAL, Rhapsody, Visual State,.....
  - **Correct by construction engineering practices:** Component Frameworks like CORBA, EJB, COMDES,.....
- Idea is to capture the semantics of these specifications as a Logic Program (LP)



Linear Hybrid Automata (LHA)

- Hybrid Model: State Machine + Continuous Dynamics
- It has:
  - locations and transitions (like state machine)
  - invariants on locations: linear (AX<= b)
  - guards on transitions : also linear
  - continuous dynamics: time invariant (dx/dt = k)



LHA Semantics

- A *Linear Hybrid Automata H = {Loc, Var, Init, Trans,D,Inv} where:*
  - ***Loc**: set of locations*
  - ***Var**: set of Variables*
  - ***Init**: assigns to each location 'l' a linear system **init(l)***
  - ***Trans**: set of transitions, where*
    - *τ= (l,γ,α,l')*
    - *γ: guard, which is a linear system over **Var***
    - *α: action, which is a linear assignment to **Var***
    - ***A transition is enabled in (l,X), if and only if γ(X) holds, and the state (l,α(X)) is then the successor of (l,X) via τ.***
  - *A labelling function D which assigns to each location **l** a linear system D(**l**) constraining variable derivatives*
  - *A labelling function **Inv, which assigns to each location l a linear system Inv(l) constraining variables***

LHA Semantics.. (2)

- State of the automaton **(l,X)**
  l:current location **X**:valuation of the variables
- State can change in two ways:
  - an enabled discrete transition changing both the control location and current valuation as imposed by α_τ
  - a *time delay, as per the variables' dynamics defined by the derivative vector X' = X + δ.dx/dt, X' satisfies the invariant associated with the location*
- A run of the automaton is an infinite sequence of states s_i:

$$s_0 \xrightarrow[\dot{X}_0]{t_0} s_1 \xrightarrow[\dot{X}_1]{t_1} s_2 \xrightarrow[\dot{X}_2]{t_2} \cdots$$

$s_i = (l_i, X_i)$ $t_i >= 0$ $t \in (0, t_i)$ $(X_i + t.\dot{X}_i)$ satisfying $Inv(l_i)$

transition successor of $s_i' = (l_i, X_i + \dot{t}_i.X_i)$ is s_{i+1}

Translation into a Logic Program

- State in LHA $s_i = (l_i,X)$ is modelled as a list $[l_i,x_1,x_2,...,x_n]$, where $x_1,x_2,...,x_n$ are components of the state vector X
- Evolution and transition behaviours are encoded a LP/driver:

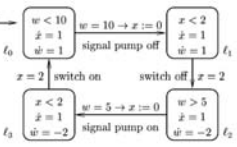
```
rState(S2) :-
        transition(S1,S2),
        rState(S1).
rState(S0) :-
        init(S0).
```


General structure of transitions

```
transition(Xs0,Xs1) :-                  % delay
        locationOf(Xs0,L0),
        before(Xs0,Xs1),
        d(Xs0,Xs1),                     %rate of change constraint
        invariant(L0,Xs1),              %invariant constraint
        d(Xs0,Xs2),                     % rate of change constraint
        before(Xs1,Xs2),
        gamma(L0,Xs2).
transition(Xs0,Xs3) :-                  % discrete
        locationOf(Xs0,L0),
        before(Xs0,Xs1),
        d(Xs0,Xs1),                     % rate of change constraint
        invariant(L0,Xs1),              % invariant
        before(Xs1,Xs2),
        d(Xs0,Xs2),                     % rate of change constraint
        gamma(L0,Xs2),                  % transition constraint
        alpha(L0,Xs2,Xs3).              % action constraint
```


LHA Specification of a Control System

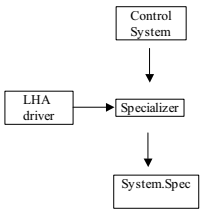
```
• init([l0,_,1,0]).
      invariant(l0,[l0,_,W1,_]) :- W1 < 1
      invariant(l1,[l1,X1,_,_]) :- X1 < 2
      invariant(l2,[l2,_,W1,_]) :- W1 > 5
      invariant(l3,[l3,X1,_,_]) :- X1 < 2
• gamma(l0,[l1,_,W1,_]) :-       W1 = 10.
  gamma(l1,[l2,X1,_,_]) :-       X1 = 2.
      gamma(l2,[l3,_,W1,_]) :-   W1 = 5.
      gamma(l3,[l0,X1,_,_]) :-   X1 = 2.
• d([l0,X,W,T],[_,X1,W1,T1]) :-  W1 is W+T1-T,    X1 is X+T1-T.
  d([l1,X,W,T],[_,X1,W1,T1]) :-  W1 is W+T1-T,    X1 is X+T1-T.
  d([l2,X,W,T],[_,X1,W1,T1]) :-  W1 is W-2*(T1-T), X1 is X+T1-T.
  d([l3,X,W,T],[_,X1,W1,T1]) :-  W1 is W-2*(T1-T), X1 is X+T1-T.
• alpha(l0,[l1,_,W1,_],[l1,X2,W2,0]) :- X2 is 0, W2 = W1.
  alpha(l1,[l2,X1,W1,_],[l2,X2,W2,0]) :- X2 is X1, W2 = W1.
      alpha(l2,[l3,_,W1,_],[l3,X2,W2,0]) :- X2 is 0, W2 = W1.
      alpha(l3,[l0,X1,W1,_],[l0,X2,W2,0]) :- X2 is X1, W2 = W1.
• before([_,_,_,T],[_,_,_,T1]) :-T < T1.
```



Specialization of LHA Driver

```
/* portion of specialised system */

/* rState__1(A,B,C,10) :- rState__1__1(A,B,C). */
rState__1__1(A,B,C) :-
        D<A,
        B is E+A-D,
        C is F+A-D,
        B<10,
        G is E+H-D,
        _ is F+H-D,
        A<H,
        G=10,
        rState__1__1(D,E,F).
rState__1__1(0,A,B) :-
        C<D,
        _ is E-2*(D-C),
        F is G+D-C,
        F<2,
        D<H,
        I is E-2*(H-C),
        J is G+H-C,
        J=2,
        B is J,
        A=I,
        rState__1__4(C,E,G).
rState__1__1(0,1,_).
```



Analysis of the Models

- Convex polyhedron analysis [Cousot & Halbwachs 78]: An abstract interpretation technique
  - Each *n-ary predicate is approximated by n-dimensional polyhedron deriving linear relationships between the arguments*
  - Safety properties boil down to proving invariants on certain system variables
    - Subjecting *system.spec* program to Convex Hull Analysis we get constraints on the predicate variables
    - whose solutions gives the invariants necessary for safety analysis
- Can generate inputs for existing model checkers

Waterlevel monitor example

Convex polyhedral tool for CLP programs (available online at http://wagner.ruc.dk/CHA/)

In this example, standard widening, and two narrowing interations.

```
w__2__5(A) :- [-1*A> -10,1*A>=1]          % level at location 0
w__3__6(A) :- [-1*A> -12,1*A>=10]         % level at location 1
w__4__7(A) :- [-1*A>= -12,1*A>5]          % level at location 2
w__5__8(A) :- [-1*A>= -5,1*A>1]           % level at location 3
```

Work in progress

- Build a tool - multiple interfaces for different formalisms
  - enabling automated translation to LP
  - single internal logic representation
- Multiple back-ends for different LP-based analyses
  - other backend possibilities
    - abstractions based on regular types
    - analysis using a greatest fixpoint model interpretation (using a proof procedure developed by Gupta)
    - a model checker

Architecture-aware Partial-order Reduction to Accelerate Model Checking of Networked Programs

Cyrille Artho, Yoshinori Tanabe, Etsuya Shibayama
National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Watcharin Leungwattanakit, Masami Hagiya
University of Tokyo, Tokyo, Japan

Programs are often structured into a main thread that delegates incoming requests, and worker threads. A similar structure also exists in applications where several processes have been merged ("centralized") into a single application. Such a transformation wraps processes as threads. and is used to model check networked programs. A direct implementation of wrapping allows for interleavings between initialization and execution of client threads. We present a partial-order reduction which, when applies to such programs, eliminates exploration of such interleavings. —

Most software model checkers [4] cannot handle multiple processes. To model check multiple processes in a single-process model checker, *centralization* has been proposed [3]. Centralization wraps several processes in a single process. Using a TCP/IP model library, networked applications can then be model checked [1]. However, the large number of thread interleavings limits scalability. Therefore, it is useful to optimize state space search as far as possible.

After centralization of an application, wrapper code runs as the main thread. The wrapper first starts the server process as a separate thread, and waits for its initialization to complete. After that, *initialization* and *execution* of each client is performed. This creates possible interleavings: After the first client is ready, it may already execute, even though the main (wrapper) thread is still initializing other clients. The model checker may analyze such interleavings, even though initialization of clients (in the main thread) does not interfere with execution of other clients. In simple programs, the model checker recognizes the redundancy in these interleavings. For more complex cases, the built-in partial order reduction fails. This observation led to a custom partial-order reduction. It takes this architectural property into account by only allowing schedules where the main (wrapper) thread finishes before client threads execute.

Using JPF version 3 [4] on small centralized programs [1], the gains achieved were not significant, because few client threads are used. However, in a more recent case study based on a different approach to analyzing networked software [2], a more complex client was analyzed. In that case, our manual optimization resulted in a significant speed-up. More work remains to be done whether centralized applications can be accelerated as well in some cases.

In the talk, reachability-based partial-order reduction in JPF is introduced first. It works on top of garbage collection. Second, custom partial-order reductions will be explained. They can be implemented either through program instrumentation or by extending the default search algorithm.

References

[1] C. Artho and P. Garoche. Accurate centralization for applying model checking on networked applications. In *Proc. ASE 2006*, Tokyo, Japan, 2006.

[2] C. Artho, B. Zweimüller, A. Biere, E. Shibayama, and S. Honiden. Efficient model checking of applications with input/output. *Post-proceedings of Eurocast 2007*, 2007. To be published.

[3] S. Stoller and Y. Liu. Transformations for model checking distributed Java programs. In *Proc. SPIN 2001*, volume 2057 of *LNCS*. Springer, 2001.

[4] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

Tutorial on Modeling VAT rules using OWL-DL

Morten Ib Nielsen, Jakob Grue Simonsen and Ken Friis Larsen
Department of Computer Science
University of Copenhagen
Email: {mortenib|simonsen|kflarsen}@diku.dk

August 28, 2007

Abstract

This paper reports on *work in progress*. We present a methodology for constructing an OWL-DL model of a subset of Danish VAT rules. It is our intention that domain experts without training in formal modeling or computer science should be able to create and maintain the model using our methodology. In an ERP setting such a model could reduce the Total Cost of Ownership (TCO) and increase the quality of the system. We have selected OWL-DL because we believe that description logic is suited for modeling VAT rules due to the decidability of important inference problems that are key to the way we plan to use the model and because OWL-DL is relatively intuitive to use.

1 Introduction

Imagine an ERP system where domain experts can create and implement changes in e.g. VAT rules without the help of programmers. The benefits would be shorter development time and fewer mistakes due to misinterpretation of specifications which lead to reduced TCO and increased quality of the software. On a coarse-grained scale such a system consists of three parts: A model of the rules, a tool to edit the model and the core ERP system using the model. In this paper we focus on the first part - the model. A priori two requirements exist. First the modeling language must be strong enough to express the rules in question and second it must be easy to use without training in formal modeling or computer science. In a more general setting the model can be used as a VAT knowledge system which external programs can query through an interface. In the long run we envision that authorities such as SKAT (Danish tax administration) can provide online access to the model e.g. using web services such that applications always use the newest version of the model.

In this paper we describe a methodology we have used to develop a model of a subset of Danish VAT rules using the general purpose Web Ontology Language (OWL) editor Protégé-OWL[1] and we report on our experiences in doing so. We selected a subset of Danish VAT rules consisting of flat VAT (25%) plus a set of exceptions where goods and services are free of VAT, chosen because they seem representative. Further the rules are accessible to us by way of an official guideline by the Danish tax administration. Our study is focusing on the feasibility

[1] `http://protege.stanford.edu/overview/protege-owl.html`.

of using OWL to model VAT rules and not on the usability of the Protégé-OWL tool itself. By feasibility we mean how easy or difficult it is (for a human) to express and understand VAT rules in OWL, in particular this does not cover issues such as modularization. The methodology presented here is inspired by the article [1] together with our own experience. Readers of this guide are assumed to have user experience of Protégé-OWL corresponding to [2] but not of computer science nor of modeling in general.

1.1 Motivation

One of the overall goals of the strategic research project 3gERP is to reduce the TCO of Enterprise Resource Planning (ERP) systems. We believe that a VAT model helps to this end in two ways. First we envision that domain experts create and update the model thus eliminating a layer of interpretation (the programmer) where errors can be introduced. Second a VAT model can change handling of VAT from being a customization task into being a configuration task, meaning that no code needs to be changed when the model is updated.

VAT and legal rules in general deal with frequent transactions between legal entities. Transactions are typically triggered when certain conditions are fulfilled and therefore dynamic checks on these conditions are needed. The idea is to use the model to automatically infer what actions should be taken based on the conditions. In the case of VAT rules we can ask the model whether a delivery is subject to VAT or not based on the information we know about the delivery. The answer from the model will be *Yes*, *No* or *Maybe*[2] and can be used to trigger an appropriate transaction. In a broader perspective the model is supposed to work as a VAT knowledge system that given a context and a question can tell other systems what to do, e.g. guide accounting systems and if required indicate that authorities should be contacted etc.

1.2 Roadmap

The remainder of this paper is structured as follows. In Section 2 we give a short account of description logic and OWL. In Section 3, 4 and 5 we present our methodology by giving examples. Finally we outline future work in Section 6 and we conclude in Section 7.

2 Description Logic and OWL

In this section we give a short introduction to description logic (DL) and OWL. This introduction can be skipped, if you are already familiar with the concepts. Description logics are knowledge representation languages that can be used to structure terminological knowledge in knowledge systems which are formally well-understood. A knowledge system typically consists of a knowledge base together with a reasoning service. The knowledge base is often split into a set of concept axioms the *TBox*, a set of assertions the *Abox* and a *Role hierarchy*. These constitute the *explicit* knowledge in the knowledge system. The reasoning service is a program that can check the consistency of the knowledge base and make implicit knowledge explicit, e.g. decide equivalence of concepts. Since the reasoning service is a pluggable component knowledge systems separate the technical task of reasoning from the problem of constructing the knowledge base.

[2] In the case where insufficient information is provided in order to answer the question.

2.1 OWL

OWL which is short for Web Ontology Language is an ontology language designed to be compatible with the World Wide Web and the Semantic Web. The most important abstraction in OWL is concept axioms which are called classes. Each class has a list of necessary conditions and zero or more equivalent lists of necessary and sufficient conditions [2]. A list of necessary conditions is a list of conditions that every member of the class must satisfy. In the same way a list of necessary and sufficient conditions is a list of conditions that must be satisfied by every member of the class and if satisfied guarantees membership in the class. OWL is based on XML, RDF and RDF-S and can be used to represent information in a way that is more accessible to applications than traditional web pages. In addition OWL has a formal semantics, which enables logic reasoning. OWL comes in three variants: OWL-Lite $\subseteq$ OWL-DL $\subseteq$ OWL-Full of increasing expressive power. The variants OWL-Lite and OWL-DL are based on the description logics $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$ respectively [3], which guarantees that important inference problems such as satisfiability and subsumption are decidable. Since OWL is XML based we need an editor to create OWL ontologies. We have used the general purpose OWL editor Protégé developed by Stanford Medical Informatics at the Stanford University School of Medicine.

3 VAT Exemption 1: Sales outside EU

Our methodology is aimed at modeling VAT rules as described in guidelines instead of the raw law text itself. This choice was made because guidelines are more accessible to us, and because these are the rules that small companies adhere to in practice. Further the investigation of the feasibility of using OWL to model VAT rules concerns the ease with which rules can be formalized and not so much from where the rules are extracted[3]. In what follows we refer to the guideline as the *legal source*.

In order to ease reading we have used the word *concept* only when we speak about the legal source. The corresponding concept in the model (OWL) is called a *class*. A concept in the legal source is modeled as one or more classes in the model.

Here we present the steps we took in order to make our model of Danish VAT rules.

3.1 Pre-modeling

1. Download Protégé-OWL from `http://protege.stanford.edu/download/release/full/` and install. Make sure you can start Protégé in OWL-mode (logic view). When started and if you select the *Class* tab it should look like Figure 1.

2. Download [2] and read it. This is **important** because many of the constructions we use are explained herein.

3.2 Modeling

First you must decide which legal source(s) you want to model.

[3]Since we have used the official guidelines by SKAT (Danish tax administration) we believe that the content of the guidelines is in accordance with the law.

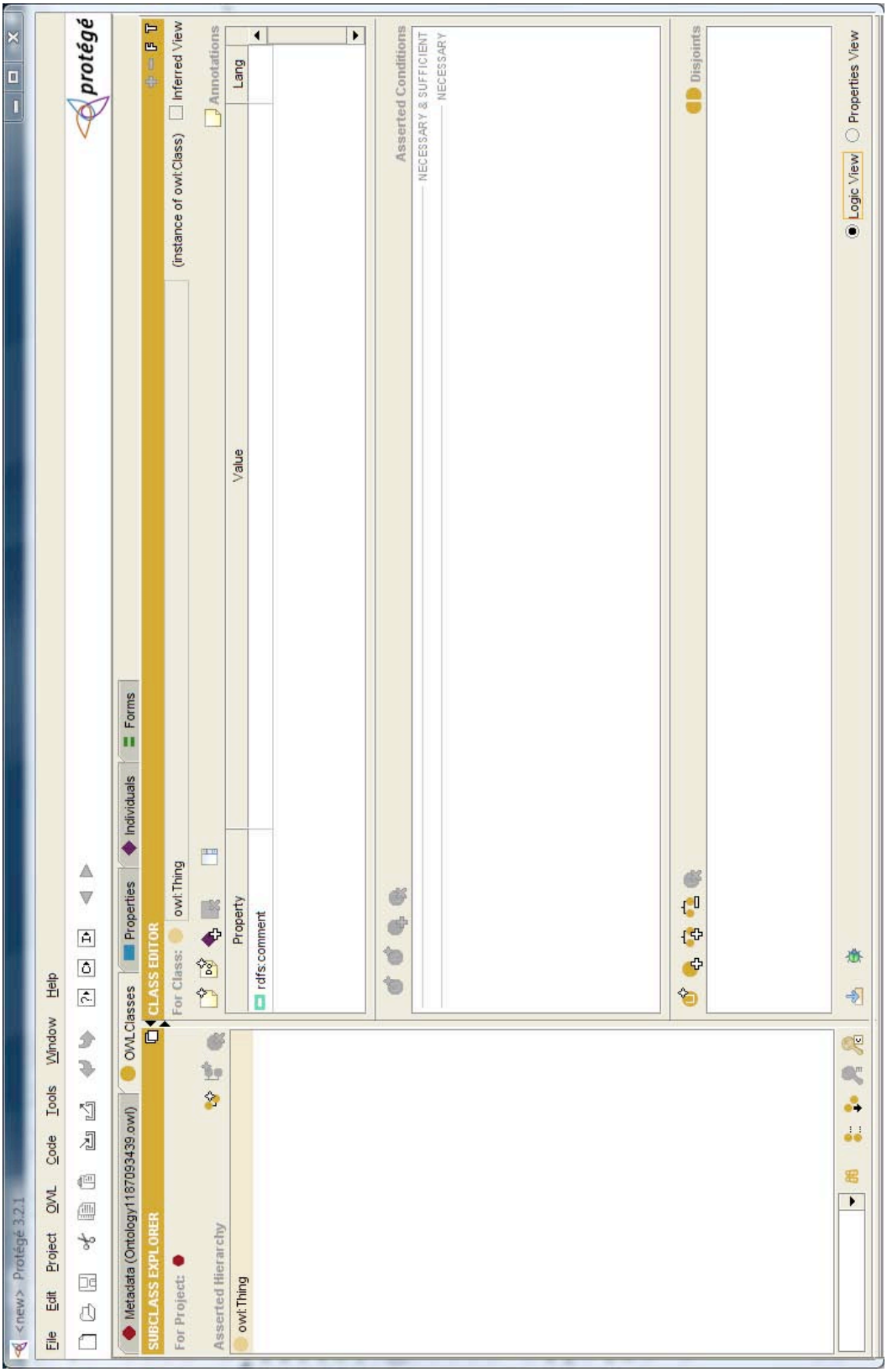


Figure 1: Protégé-OWL class-tab, logic view.

In our case we used the official guideline *Moms - fakturering, regnskab mv, E nr. 27, Version 5.2 digital, 19. januar 2005.*

3.2.1 Overall framework

Modeling should start with a read through of the legal source. Based on this general (to be refined later) classes such as `Location`, `Goods`, `Services` and `FreeOfVAT` together with attributes such as `hasDeliveryType` and `hasSalesPrice` can be created as subclasses of the built-in top-level class `owl:Thing`. An attribute can usually take on at most a finite number of values. In that case we use value partitions to model them as described in [2][p. 73-76][4]. If the domain is not finite we use data type properties instead. Deciding on the overall framework helps to structure the capturing of rules in a homogeneous way and enables working in parallel (which can be needed if the legal source is large). After our read through of the legal source we arrived at the overall framework in Figure 2.

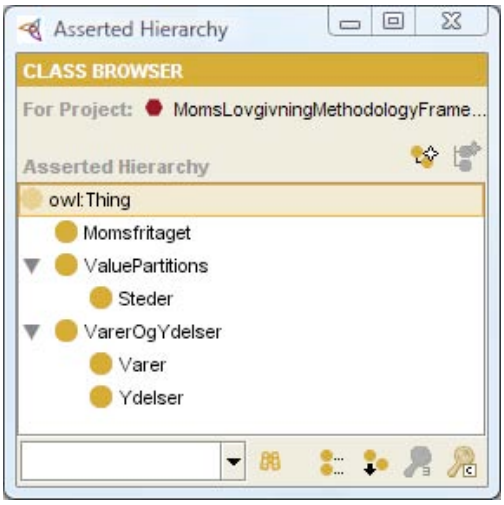


Figure 2: Overall framework.

`Naming Convention.` All classes, properties, individuals etc. should be given names picked from or inspired of the legal source. All names should be in the same language as the legal source (in our case Danish). Using the naming convention supported by Protégé-OWL class and individual names should be written in Pascal Notation, e.g. *InternationalOrganization* not *internationalOrganization* or *International_Organization*, while property names are written in Camel Hump Notation, e.g. *someProperty*. Typically a property is used to assign an attribute to a class. In this case we prefix the name of the property with a verb describing the kind of relation the class has along that property, e.g. *hasNumberOfSides* or *isFragile*.

3.2.2 Rule modeling - step I

Having modeled the overall framework it is time to go through the legal source one section at a time looking for rules that should be modeled. Here we give an elaborate description of how to model a single rule from the legal source starting from the overall framework in Figure

[4] An exception is the domain of truth values, which is built-in as a data type.

Table 1 Extract from the legal source and its translation into English.

Salg til lande uden for EU (3. lande). Du skal ikke beregne moms af varer, du leverer til steder udenfor EU eller til Færøerne og Grønland. Det samme gælder normalt også for ydelser, men du skal dog opkræve moms af visse ydelser.

[4][p. 9]

And translated into English:

Sales outside EU (3rd countries). No VAT should be added to goods delivered to destinations outside the European Union, or to the Faroe Islands or Greenland. This fact ordinarily also applies to services, but VAT should be added to certain services.

Translated from [4][p. 9]

Table 2 Necessary & sufficient conditions for application of the rule in Table 1.

- The rule concerns sales.
- The rule concerns both goods and services.
- The place of delivery must be outside the European Union, or the Faroe Islands or Greenland.

2. In Section 4 and 5 we give a brief description of how to model other rules. Together the modeling of these rules cover all the constructions we have used in our VAT model. Since our legal source is in Danish we present the rules in their original Danish phrasing together with a translation into English. Now let us consider the rule shown in Table 1.

Since our model is only a prototype we make a slight simplification and assume that the rule also applies to *all* services. With this simplification we can identify the necessary and sufficient conditions for application of the rule. These are shown in Table 2. In order to model the necessary and sufficient conditions in Table 2 we must add some attributes to `VarerOgYdelser`. The first and second condition in Table 2 tell us that we must be able to model that goods and services are sold[5]. We do that by adding an attribute to the class `VarerOgYdelser` (translates into `GoodsAndServices`) which already exists in our overall framework. Attributes are modeled using functional properties. In accordance with our naming convention we select the name `harLeveranceType` (translates into `hasDeliveryType`). Since there is a finite number of delivery types we model this attribute as a *value partition*, i.e. an enumeration. Value partitions can be created using a built-in wizard[6]. Just as in [2] we store value partitions as subclasses of the class `ValuePartitions`. The reason plain enumerations are not used is that they cannot be sub-partitioned. Using value partitions we retain the possibility of further refining the concepts the value partitions model.

[5]Instead of being sold goods can also be used as e.g. a trade sample. See [4][p. 8-9] for other examples.
[6]Menu►Tools►Patterns►Value Partition....

Remark. Technically enumerations are constructed by defining a class in terms of a finite set of individuals plus a functional property that has this class as its range. Since individuals are atoms they cannot be subdivided. On the other hand a value partition is defined using a functional property having as its range a class defined as the union of its subclasses all of which are distinct. These subclasses can (because they are classes) be partitioned into more subclasses if needed.

Having created the value partition `harLeveranceType` which can have `Salg` (translates into `Sale`) as a value we need to add it as an attribute to the class `VarerOgYdelser`. This is done by adding to the *necessary conditions* an existential quantification over the corresponding property having the value partition (or data type in case of data type attribute) as its range. Thus we add ∃ `harLeveranceType some LeveranceType` to `VarerOgYdelser`. The third condition tells us that we must be able to model that goods and services have a place of delivery. A read through of the legal source tells us that only three places are needed namely *Denmark*, *EU* and *non-EU*. Thus this attribute which we name `harLeveranceSted` (translates into `hasPlaceOfDelivery`) must be modeled as a value partition.

Having modeled these attributes the class `VarerOgYdelser` looks as shown in Figure 3.

3.2.3 Rule modeling - step II

Now we are ready to model the rule itself. Since the rule describes a situation where you do not have to pay VAT we model it as a subclass of `Momsfritaget` (translates into `FreeOfVAT`). Following our naming convention we name the class `MomsfritagetSalgAfVarerOgYdelserTilIkke-EU` (translates into `VATFreeSalesOfGoodsAndServicesInNon-EU`). Then we add a textual description of the rule and a reference to where in the legal source the rule stems from to the `rdfs:comment` field. Next we must specify *necessary and sufficient* conditions on membership in `MomsfritagetSalgAfVarerOgYdelserTilIkke-EU`. It is important to remember that if a class has two sets of necessary and sufficient conditions then they must imply each other, see [2][p. 98]. Based on the necessary and sufficient conditions captured in Table 2 we add the following necessary and sufficient conditions to `MomsfritagetSalgAfVarerOgYdelserTilIkke-EU`:

- `VarerOgYdelser`
- ∃ `harLeveranceSted some Ikke-EU`
- ∃ `harLeveranceType some Salg`

The result is shown in Figure 4.

4 VAT Exemption 2: Sales to Embassies

In this section and onwards we will not mention when to add references to the legal source in `rdfs:comment` fields of classes and properties. The rule of thumb is that this should always be done. Now let us consider the rule in Table 3. We identify the necessary and sufficient conditions for application of the rule. These are shown in Table 4.

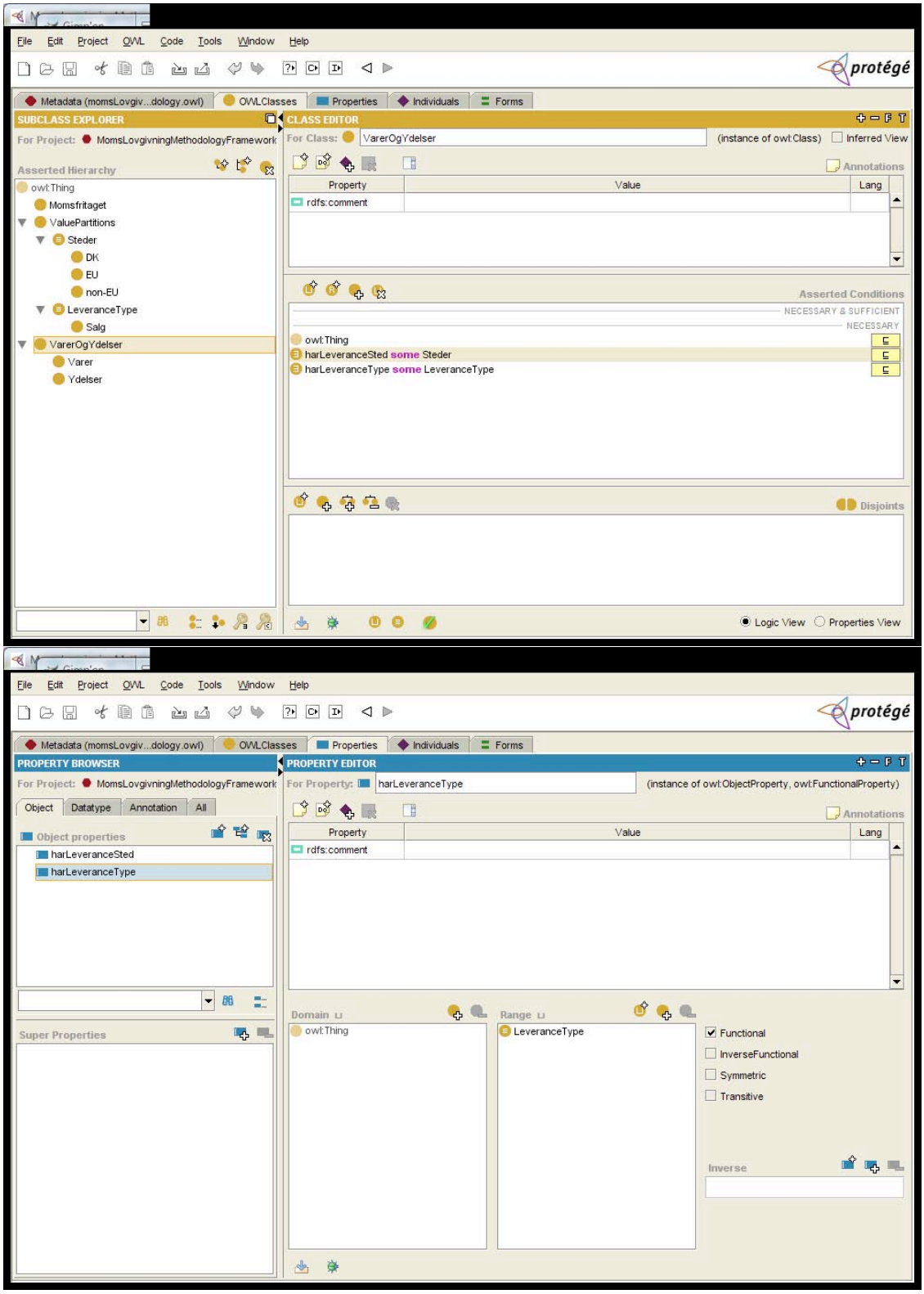


Figure 3: Class and property view after adding attributes.

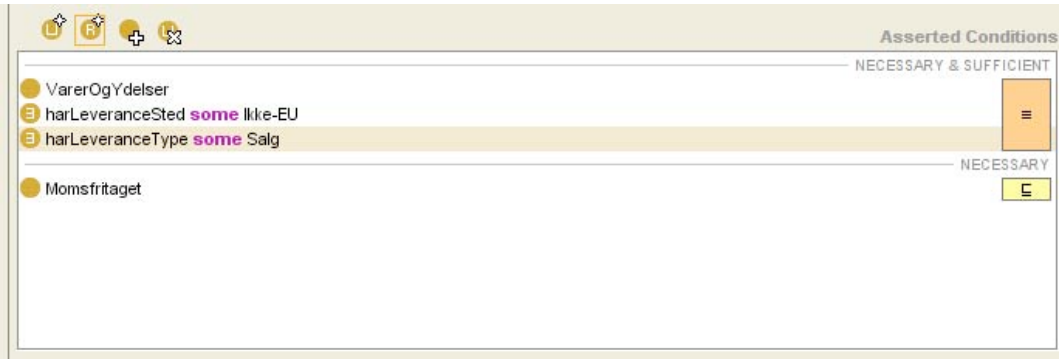


Figure 4: Asserted Conditions of our model of the legal rule in Table 1.

Table 3 Extract from the legal source and its translation into English.

> *Salg til ambassader. Du skal ikke beregne moms af varer og transportydelser, som du leverer til ambassader og internationale organisationer i andre EU-lande.*
>
> [4][p. 9]

And translated into English:

> *Sales to embassies. VAT should not be added to goods and transport services delivered to embassies and international organizations in countries within the European Union.*
>
> Translated from [4][p. 9]

Table 4 Necessary & Sufficient conditions for application of the rule in Table 3.

- The rule concerns sales.
- The rule concerns goods and transport services.
- The place of delivery must be in the European Union.
- The buyer must be an embassy or an international organization.

4.1 Rule modeling - step I

We are already able to model that the rule concerns sale and that the place of delivery must be in EU. We cannot model the specific service transportation yet. Therefore we must add it to our model. Since it is a service it should be modeled as a subclass of `Services`. We name the class modeling the service transportation `Transport` (translates into `Transportation`). Now we can model that something belongs to the set of goods and transport services by requiring membership of `Varer` ⊔ `Transport`. Finally we must be able to model that the buyer is an embassy or an international organization. Since there are only finitely many different kinds of buyers we model this as a value partition, and because this attribute applies to both `Varer` and `Transport` we add it to their most specific common super-class which is `VarerOgYdelser`. We name this attribute `harKøberType` (translates into `hasKindOfBuyer`). After having done all this the model looks as shown in Figure 5.

4.2 Rule modeling - step II

Having added all the necessary classes and attributes to the model we are ready to model the rule itself. Since the rule describes a situation where you do not have to pay VAT we model it as a subclass of `Momsfritaget`. Following our naming convention we name the class `MomsfritagetSalgTilAmbassaderOgInternationaleOrganisationerIEU` (translates into `VATFreeSalesToEmbassiesAndInternationalOrganizationsInEU`). Based on the necessary and sufficient conditions captured in Table 4 we add the following necessary and sufficient conditions to `MomsfritagetSalgTilAmbassaderOgInternationaleOrganisationerIEU`:

- `harLeveranceType some Salg`
- `Varer` ⊔ `Transport`
- `harLeveranceSted some EU`
- `harKøberType some AmbassadeOgPersonaleMedDiplomatiskeRettigheder`

The result is shown in Figure 6.

5 VAT Exemption 3: Sales in other EU countries

In this section we consider one final rule, the rule in Table 5. We identify the necessary and sufficient conditions for application of the rule. These are shown in Table 6.

5.1 Rule modeling - step I

We are already able to model that the rule concerns sale of goods delivered inside the European Union. The new thing is that we must be able to indicate whether a buyer is registered for VAT and if so, we must register the buyers VAT registration number. We use a functional data type property named `erKøberMomsregistreret` (translates into `isTheBuyerRegisteredForVAT`) with the data type `xsd:boolean` as its range to model whether the buyer is registered for VAT. Similarly we use a functional data type property named `erKøbersMomsnummer` (translates into `isBuyersVATRegistrationNumber`) with the data type `xsd:string` as its range to register the buyers VAT registration number if he has one.

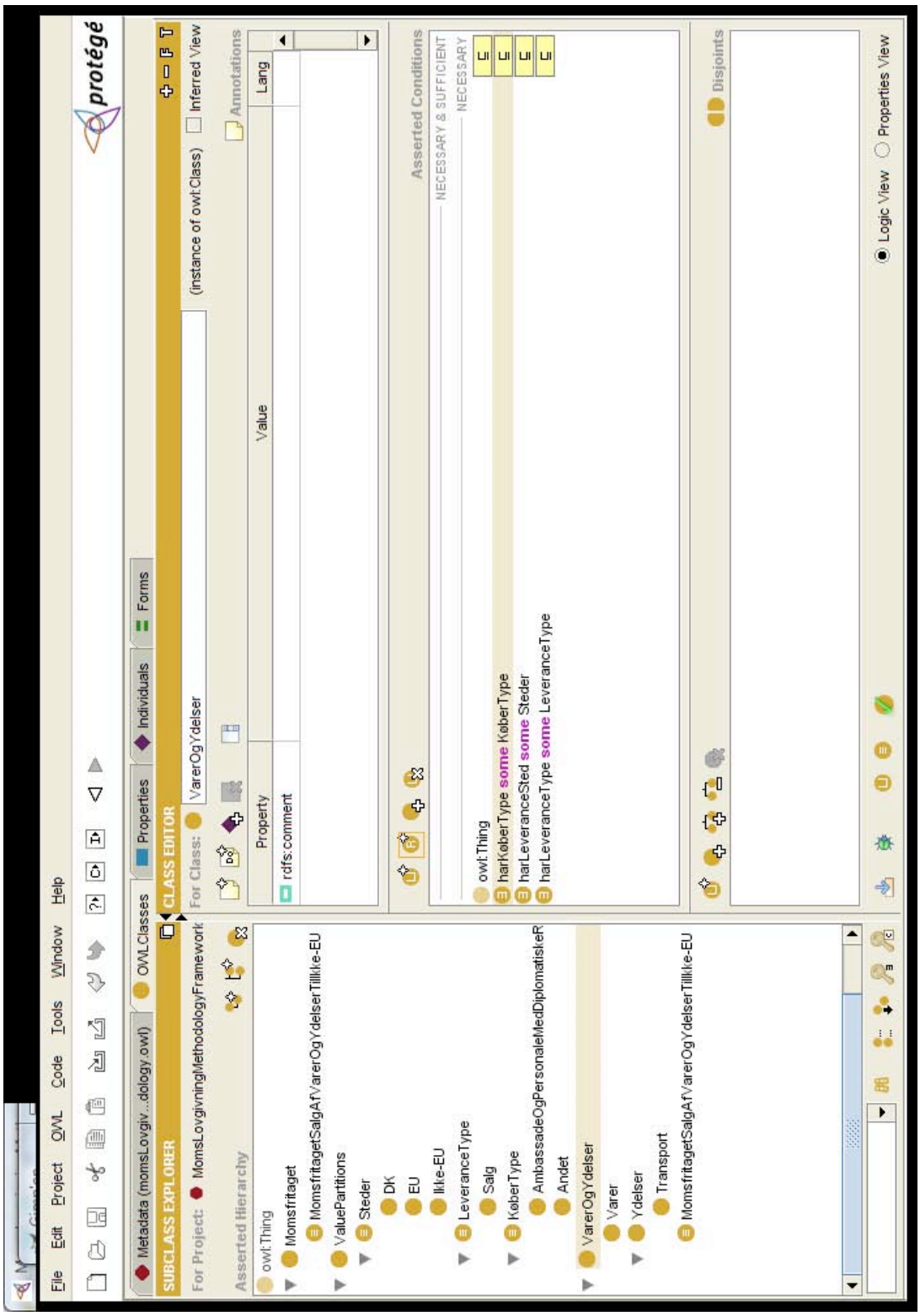


Figure 5: The model after adding classes and attributes as described in Section 4.1.

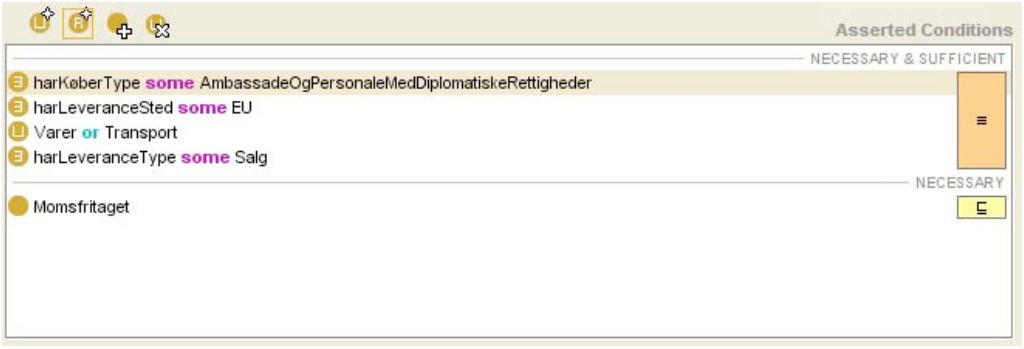


Figure 6: Asserted Conditions of our model of the legal rule in Table 3.

Table 5 Extract from the legal source and its translation into English.

Salg til andre EU-lande. Du skal ikke beregne dansk moms, når du sælger varer til momsregistrerede virksomheder i andre EU-lande. Du skal derfor sørge for at få virksomhedens momsnummer.

[4][p. 8]

And translated into English:

Sales in other EU countries. No VAT should be added to goods delivered to companies in other EU countries, provided that the companies are registered for VAT. In this case you must acquire the VAT registration number of the company.

Translated from [4][p. 8]

Table 6 Necessary & Sufficient conditions for application of the rule in Table 5.

- The rule concerns sales.
- The rule concerns goods.
- The place of delivery must be in the European Union.
- The buyer must be registered for VAT.
- You must acquire the VAT registration number of the company.

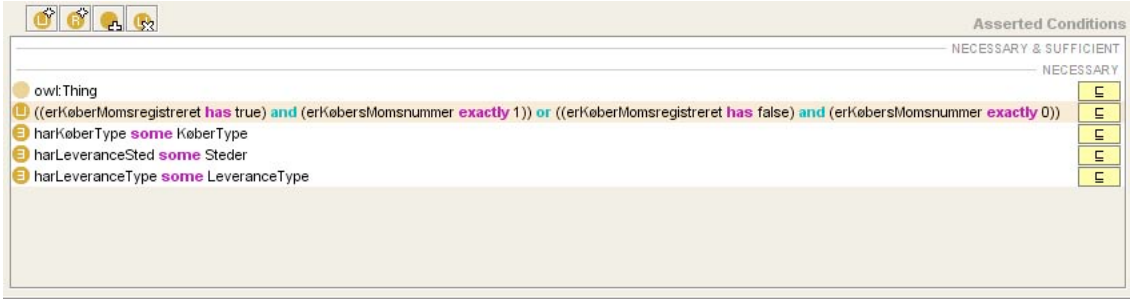


Figure 7: Asserted Conditions of `VarerOgYdelser` after adding the requirement for registering VAT registration numbers.

A read through of [4] will reveal that you must register the VAT registration number of the buyer exactly when the buyer is registered for VAT. Thus we model this as a property of `VarerOgYdelser` and not of `Varer` (as indicated by the rule). The requirement can be modeled as follows:

- `((erKøberMomsregistreret has true)` $\sqcap$ `(erKøbersMomsnummer exactly 1))` $\sqcup$ `((erKøberMomsregistreret has false)` $\sqcap$ `(erKøbersMomsnummer exactly 0))`

The result is shown in Figure 7.

5.2 Rule modeling - step II

Having added the necessary attributes to the model we are ready to model the rule itself. Since the rule describes a situation where you do not have to pay VAT we model it as a subclass of `Momsfritaget`. Following our naming convention we name the class `MomsfritagetSalgTilAndreEU-lande` (translates into `VATFreeSalesToOtherEUCountries`). Based on the necessary and sufficient conditions captured in Table 6 we add the following necessary and sufficient conditions to `MomsfritagetSalgTilAndreEU-lande`:

- `harLeveranceType some Salg`
- `Varer`
- `harLeveranceSted some EU`
- `erKøberMomsregistreret has true`

We note that the obligation to register the buyers VAT registration number is modeled indirectly, see Section 5.1. The result is shown in Figure 8.

6 Future work

Since this is work in progress there are a lot of areas we need to address. In the near future we plan to integrate our model in a prototype ERP system as described in the introduction. This opens the posibility for modeling the parts of the Danish VAT legislation concerning depreciation and VAT reporting (since they are intertwined and contain a lot of technical

Figure 8: Asserted Conditions of our model of the legal rule in Table 5.

requirements on the financial reports). We also need to model other countries VAT rules in order to confirm that Danish VAT rules are indeed representative with respect to the constructions that are needed in the modeling language. Based on this we need to refine our overall framework such that it captures the common structure and we need to identify what kinds of questions a model must be able to answer. The synthesized knowledge from modeling the VAT rules of other countries should also result in a more detailed analysis of what we can and cannot model.

Based on all this we should design a minimal description logic extended with the needed functionality identified in the analysis just mentioned, such as predicates like $x < 100$ which are needed in some rules. We should also provide a reasoner for the logic together with an editor such that the above process can be repeated.

Finally in order to compare our OWL model with a different approach we want to make a model using Datalog, which is the de facto standard language used to express rules in deductive databases, of the rules we have formalized in OWL already. It would also be interesting to try a hybrid solution e.g. OWL plus a rule language like SWRL. This work is independent of the tasks mentioned above and can be carried out in parallel.

7 Conclusion

We have shown how to model a subset of Danish VAT rules concerning exemption from VAT using Protégé-OWL. First we created an overall framework for the VAT model with the property that legal rules and the concepts they involve can be modeled as subclasses of existing classes in the framework. This helps to ensure that related concepts are modeled in the same way and that a single concept is not modeled twice. The second step was an iterative process consisting of two steps repeated for each rule. The first step is to extend the model such that the rule in question can be modeled. This is done by modeling concepts from the legal source as classes in the model and by adding attributes to the necessary conditions of such classes. The second step is to model the rule itself. This is done by adding specific requirements for application of the rule to the necessary and sufficient conditions of the class modeling the rule.

The step by step iterative modeling has been working fine in practice and an extension to cover several different VAT and duty rates does not seem to be problematic as long as they do not require us to model restrictions such as $x < 100$ which is not supported directly in OWL [7].

[7] Whether this is a weakness of OWL, or just us trying to use OWL for something it was not designed to

Apart from modeling inequalities we have not had modeling problems. One problem though is that reasoning about individuals in OWL models is not supported very well. Therefore we have tried to avoid the use of individuals wherever possible (using value partitions).

do is not clear at the moment.

References

[1] T.J.M. Bench-Capon, F.C.: Isomorphism and legal knowledge based systems. Artificial Intelligence and Law **1**(1) (1992) 65–86

[2] M. Horridge, H. Knublauch, A.R.R.S., Wroe, C.: A practical guide to building owl ontologies using the protégé-owl plugin and co-ode tools edition 1.0. (2004)

[3] Horrocks, I., Patel-Schneider, P.F.: Reducing owl entailment to description logic satisfiability. Lecture Notes in Computer Science **2870/2003** (2003) 17–29

[4] ToldSkat: Moms - fakturering, regnskab mv. (Vejledning E nr. 27) Version 5.2. (2005)

THE SEMANTICS OF "SEMANTIC PATCHES" IN COCCINELLE

Neil D. Jones and René Rydhof Hansen
DIKU, University of Copenhagen

WHAT IS COCCINELLE?

- A ladybug (a bug that eats other bugs)
- A system to automate and document collateral evolutions in Linux device drivers
- A research project (DIKU: Julia Lawall...; Nantes: Gilles Muller...)
- Practical motivation: as in Julia Lawall's and Henrik Stuart's talks.

CONTRIBUTION OF THIS WORK

- Rational reconstruction of the semantics of the core of Coccinelle's transformation language SmPL
  - Clarify some points in the semantics
  - Compactly and explicitly describes core of a practical system
  - Executable: implemented as a functional program
- A Coccinelle novelty: to use temporal logic CTL as an intermediate language to implement SmPL.
  - (As with compiler intermediate languages, users need not know of or be aware of CTL.)
- A theoretical bridge: prove
  – the natural pattern-matching way to read SmPL
  – is equivalent to its CTL implementation.

SEMANTIC PATCHES

- Semantic Patch notation SmPL abstracts and generalises the Linux kernel community's "patch notation"
- Patches are (in effect a sort of) inverse to the `diff` utility
- Coccinelle is an executable program transformer:

  $$Coccinelle : Semantic\ patch \times Source\ text \rightarrow Target\ text$$

- Example of a semantic patch:

```
@@ expression E; @@                    [ declare pattern variable E ]
  cli();
  ... WHEN != ( sti(); | restore_flags(...); )
- usb_submit_urb(E)
+ usb_submit_urb(E, GFP_ATOMIC)
```

- Effect: adds `GFP_ATOMIC` as the second argument to `usb_submit_urb` when a lock `cli()` is held.

CONTEXT

- Coccinelle focus is not compiler optimisation, but software updating.
- Coccinelle is intentionally not semantics-preserving, in contrast to compilers or program transformers.
- The reason: Coccinelle may be used to change program functionality, or to fix or to detect bugs (Henrik Stuart's talk!).
- Coccinelle does not require familiarity with (sometimes rather subtle) temporal logic.
  – A reason: usability by a broad software engineering community,
- Instead, Coccinelle uses patterns with variables and the "..." operator to localise transformation sites.

A CORE LANGUAGE FOR SEMANTIC PATCHES

A ground term is a tree structure built from operators.

At first, a source program = a straight-line sequence of ground terms.

Later: extend to include an arbitrary control flow graph with branches, divergence, convergence and loops.

SYNTAX OF CORE-SMPL SEMANTIC PATCHES

P ::=	ε	Pattern that matches the empty sequence of terms
	EP	A match for E followed by a match for P
E ::=	T	Matches a term T
	$(P_1 \,'\|'\, P_2)$	Match P_1 or P_2
	...	Match sequence of zero, one, or more terms
	$-T$	Delete one T: match it, but do not copy it to the outp
	$+T$	Insert T in the output sequence (no matching occurs)
T ::=	x	A term is like a ground term, but may contain variables
	$op(T_1, \ldots, T_k)$	
x ::=	*variable*	A pattern variable

EXAMPLE TRANSFORMATIONS (NO VARIABLES)

1. $\mathcal{T}[\![\texttt{...abab...}]\!]$ (abcd) = $\emptyset$
2. $\mathcal{T}[\![\texttt{...abab...}]\!]$ (cababababd) = {cababababd}
3. $\mathcal{T}[\![\texttt{...a-ba-b+e+f...}]\!]$ (cababd) = {caaefd}
4. $\mathcal{T}[\![\texttt{...a-ba-b+e+f...}]\!]$ (cababgababd) = {caaefgaaefd}
5. $\mathcal{T}[\![\texttt{...a-ba-b+e+f...}]\!]$ (cababababd) = {caaefababd, cabaaefabd, cababaaefd}

Example 4:

transforms in two places

Example 5: Coccinelle system rejects this, since

multi-valued transformations are not allowed

SEMANTIC RULES (continuation $c : in \longrightarrow 2^{out}$)

I :

$\mathcal{T}[\![P]\!] = \mathcal{P}[\![P]\!]\ c_0$ where $c_0\ in = \{in\}$

II : Sequences of things

$\mathcal{P}[\![\varepsilon]\!]\ c\ in = (c\ in)$

$\mathcal{P}[\![\ E\ P\ ]\!]\ c = \mathcal{E}[\![E]\!]\ (\mathcal{P}[\![P]\!]\ c)$

III : Single things

$\mathcal{E}[\![G]\!]\ c\ [] = \emptyset$

$\mathcal{E}[\![G]\!]\ c\ (G' :: in) =$ if $G = G'$ then $\{G :: out \mid out \in (c\ in)\}$ else $\emptyset$

$\mathcal{E}[\![\ P_1 \mid P_2\ ]\!]\ c\ in = (\mathcal{P}[\![P_1]\!]\ c\ in) \cup (\mathcal{P}[\![P_2]\!]\ c\ in)$

$\mathcal{E}[\![...]\!]\ c\ in = (c\ in)\ \cup \{G :: out \mid G :: in' = in$ and $out \in (\mathcal{E}[\![...]\!]\ c\ in')\}$

IV : Deletion, insertion

$\mathcal{E}[\![-G]\!]\ c\ [] = \{\}$

$\mathcal{E}[\![-G]\!]\ c\ (G' :: in) =$ if $G = G'$ then $(c\ in)$ else $\emptyset$

$\mathcal{E}[\![+G]\!]\ c\ in = \{G :: out \mid out \in (c\ in)\}$

REMARKS ON THE SEMANTICS

1. Semantics simple, similar to regular expression matching
2. In essence a higher-order functional program (has been programmed in Haskell)
3. Easy extension to pattern variables (add environment argument)
4. A less easy extension: in practice

   a source program = a control-flow graph

   and not a straight-line sequence of terms

   Matching usually extended ad hoc in the literature
5. Also, an efficiency problem — with this implementation — non-linear use of continuation c to implement $|, \ldots$
6. To solve both problems: CTL comes to the rescue!

A TWO-STEP APPROACH: COMPILING SmPL TO CTL

Coccinelle uses CTL as a compiler intermediate language.
(This happens "under the hood": users don't have to know CTL, model checking, etc.)

- Natural extension from sequences to program control flow graphs (CFGs)
- Interaction between universal and existential quantification is well-defined in temporal logic;
  no extra work to generalise to patterns with alternating computation path quantifiers.
- Performance advantage: model checking is fast, well-engineered
- Flexibility advantage: same CTL can be used with different translation schemes for other Coccinelle applications, e.g., bug finding.

CTL FORMULAS AND PROGRAM MODELS

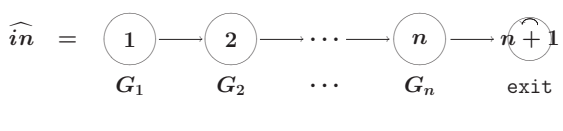
Let AP be a set of atomic propositions ap. CTL formula syntax:

$$\phi ::= ap \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid AX\phi \mid EX\phi \mid A(\phi U \phi) \mid AF(\phi)$$

Model satisfaction relation:

$$\mathcal{M}, s \models \phi$$

Model (if the source program is a straight-line term sequence $G_1 \ldots G_n$) :



(S = states)

(L = atomic properties)

More generally: the program's flow chart is a Kripke model

$$\mathcal{M} = (S, \rightarrow, L)$$

COMPILING FROM SmPL INTO CTL

$k : K = \texttt{tail} \mid \texttt{after}\ CTL$ — The "compiler's continuation" (as first-order data)

$$
\begin{array}{lcll}
\mathcal{T}_{ctl}[\![P]\!] & = & \mathcal{P}_{ctl}[\![P]\!]\ \texttt{tail} & \\
\mathcal{P}_{ctl}[\![\varepsilon]\!]\ \texttt{tail} & = & \texttt{true} & \\
\mathcal{P}_{ctl}[\![\varepsilon]\!]\ (\texttt{after}\ \phi) & = & \phi & \\
 & & & \\
\mathcal{P}_{ctl}[\![E\ P]\!]k & = & \mathcal{E}_{ctl}[\![E]\!](\texttt{after}(\mathcal{P}_{ctl}[\![P]\!]k)) & \\
 & & & \\
\mathcal{E}_{ctl}[\![G]\!]\ \texttt{tail} & = & G & \text{ground term } G \text{ is atomic proposition} \\
\mathcal{E}_{ctl}[\![G]\!]\ (\texttt{after}\ \phi) & = & G \wedge AX\phi & \\
 & & & \\
\mathcal{E}_{ctl}[\![P_1 \mid P_2]\!]k & = & (\mathcal{P}_{ctl}[\![P_1]\!]k) \vee (\mathcal{P}_{ctl}[\![P_2]\!]k) & \\
 & & & \\
\mathcal{E}_{ctl}[\![\ldots]\!]\ \texttt{tail} & = & AF\ \texttt{exit} & \text{end of the input} \\
\mathcal{E}_{ctl}[\![\ldots]\!]\ (\texttt{after}\ \phi) & = & AF\ \phi & \text{eventually a future state must satisfy } \phi
\end{array}
$$

AN EXAMPLE: COMPILING FROM SmPL INTO CTL

$$\mathcal{T}_{ctl}[\![\ldots\texttt{aba}\ldots]\!] = AF\ (a \wedge AX(b \wedge AX(a \wedge AF\ \texttt{exit})))$$

Correctness of translation from Smpl to CTL

Theorem For straight-line source program in and pattern P without $+, -$ or variables:

$$\mathcal{T}[\![P]\!]\ in = \{in\}$$

if and only if

$$\widehat{in}, 1 \models \mathcal{T}_{ctl}[\![P]\!]$$

Proof by induction on pattern syntax. Relates the
– match-time continuation (a function) to the corresponding
– compile-time continuation (a data structure).

REMARKS

- Top-down matching of a pattern against a straight-line sequence is equivalent to Bottom-up model checking of the pattern's CTL translation
- Omitted: how to extend CTL to do transformations (+ and - in patterns). Coccinelle does it during post-processing, after model checking.
- There's a close correspondence between
  - CTL's path quantifiers A, E
  - compiler practice re. "must" and "may" flow analyses

  (they just amount to "all-path" and "some-path" properties)
- Extensible to patterns containing variables, using CTL-V (CTL with variables)

CTL-V = CTL WITH PATTERN VARIABLES

CTL-V syntax: has both path and pattern quantification:

$$\phi \ ::= \ T \mid \neg\phi \mid \ldots \mid EX\phi \mid \ldots \mid \exists x\phi$$

where

- "atomic" proposition T may contain pattern variables x
- p ranges over fragments of the source program P being analysed

Extended satisfaction relation:

$s \models_\theta T$ iff some $\theta = MGU(T, v)$ where $L(s) = \{v\}$
$s \models_\theta \neg\phi$ iff not $s \models_\theta \phi$
...
$s \models_\theta EX\phi$ iff $\exists \sigma \in \mathbb{P}(s) \ . \ \sigma[1] \models_\theta \phi$
...
$s \models_\theta \exists x\phi$ iff $s \models_{\theta[x\mapsto v_1]} \phi$ or ... or $s \models_{\theta[x\mapsto v_m]}$
where

$$Val = \{v_1, \ldots, v_m\}$$

is the (finite!) set of all fragments of progam P.

CUTTING TO THE FINISH...

- We've seen a rational reconstruction.
- Not the whole truth; it abstracts a working system down to a not-too-big set of core concepts.
- Coccinelle has satisfactory expressiveness, efficiency, user-friendliness
- The CTL-V extension is enough for Coccinelle's program transformations
- An extended CTL model checker is used (described in the full paper)
- The semantics of CTL-V with variables, transformation and quantifiers is complex (current work)

RELATED WORK

Software updating: university groups at

- Nantes (Muller, Padioleau, ...),
- Copenhagen (Lawall, Hansen, Jones, ...);
- Oxford (De Moor, Lacey, ...); and
- Stony Brook (Liu, Stoller, ...).

De Moor and Liu: apply regular expressions to program transformation.

Lacey, Jones,...: compiler optimisation (semantics-preserving) by conditional rewrites

$$\boxed{\mathsf{C} \Rightarrow \mathsf{C}' \text{ if } \phi}$$

where C is a pattern, C′ its replacement. Enabling condition ϕ may refer to the computational past or future, relative to C.

Powerful but... the user must know temporal logic.

Stratego: less semantics-based but more powerful as a rewriting engine.

Vu-X : Website Updating System based on Bidirectional Transformation

Keisuke Nakano (University of Tokyo)
Supported by PSD / DXP project

The 3rd DIKU-IST Workshop, October 5, 2007.

What is "website" ?

- Website = a set of webpages

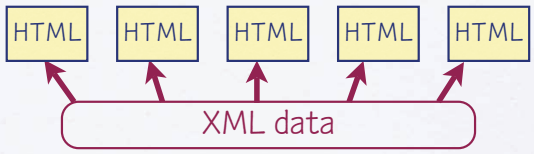
Independently-Made Website Construction

- Every webpage is independent.

  - Easy to find where should be modified
    - Where is wrong = Where should be updated
  - Hard to maintain consistency of contents
    - Some information may be shared.

Transformation-Based Website Construction

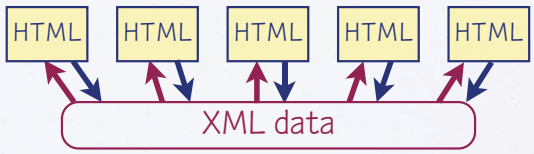
- Every webpage is automatically generated.



  - Easy to maintain consistency of contents
    - A shared information comes from XML data.
  - Hard to find where should be modified
    - Where is wrong ≠ Where should be updated

Bidirectional-Transformation-Based Website Construction

- Every transformation is bidirectional.



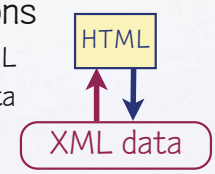
  - Easy to maintain consistency of contents
    - as with transformation-based
  - Easy to find where should be modified
    - as with independently-made

Vu-X : Website Updating System based on Bidirectional Transformation

- It is easy to maintain consistency.
- Only one-way transformation is needed.
- No special environment is needed.
- Each page can be modified on a web browser.

Bidirectional Transformation

- Pair of two-way transformations
  - Reflects a change of XML data to HTML
  - Reflects a change of HTML to XML data
- Fwd : XML → HTML
  - Extracts data and layout them
- Bwd : (XML × HTML) → XML
  - Reflects an update on HTML
  - Depends on the original XML data
- Fwd and Bwd must be consistent.

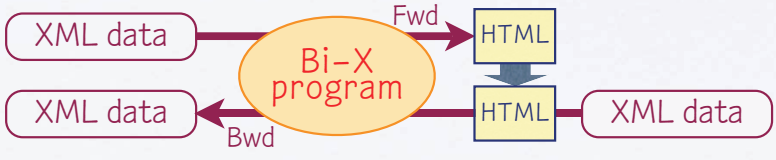


Vu-X : Website Updating System based on Bidirectional Transformation

- ☑ It is easy to maintain consistency.
  - Vu-X enjoys bidirectional transformation.
- Only one-way transformation is needed.
- No special environment is needed.
- Each page can be modified on a web browser.

Bi-X : Bidirectional Transformation Language [Liu et al. 06]

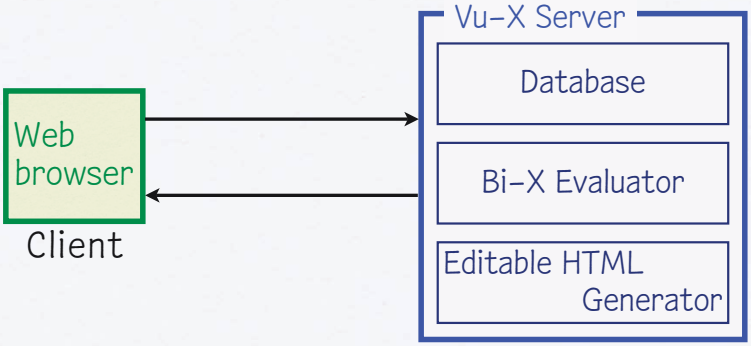
- One-way transformation is enough.
  - We just specify a single program
    - as Fwd : XML → HTML
    - but it can be used for Bwd : (XML×HTML)→XML



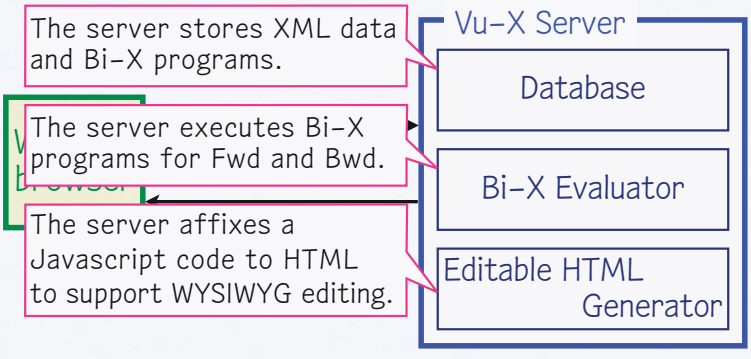
Vu-X : Website Updating System based on Bidirectional Transformation

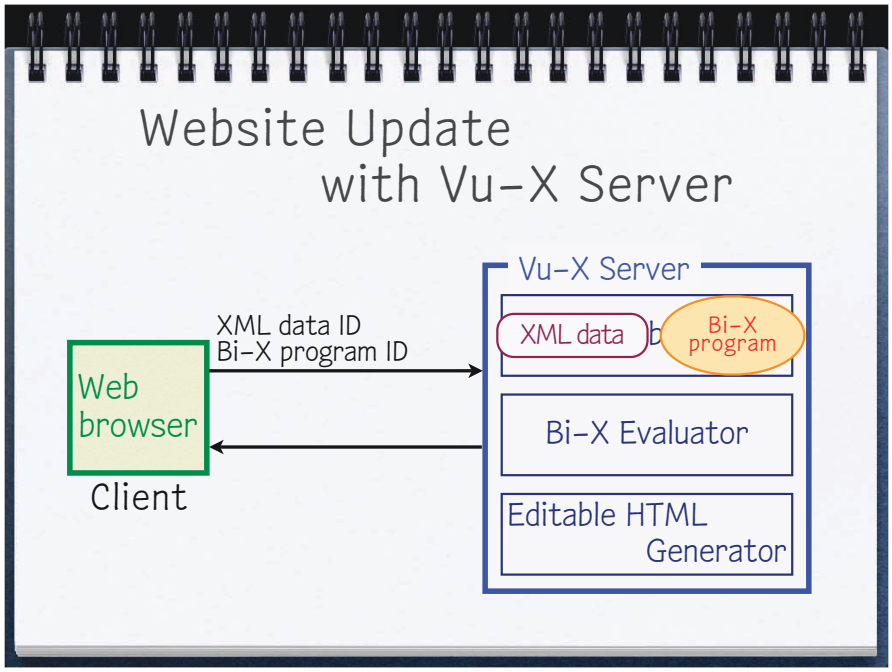
- ☑ It is easy to maintain consistency.
  - Vu-X enjoys bidirectional transformation.
- ☑ Only one-way transformation is needed.
  - Transformations are written in Bi-X language.
- ☐ No special environment is needed.
- ☐ Each page can be modified on a web browser.

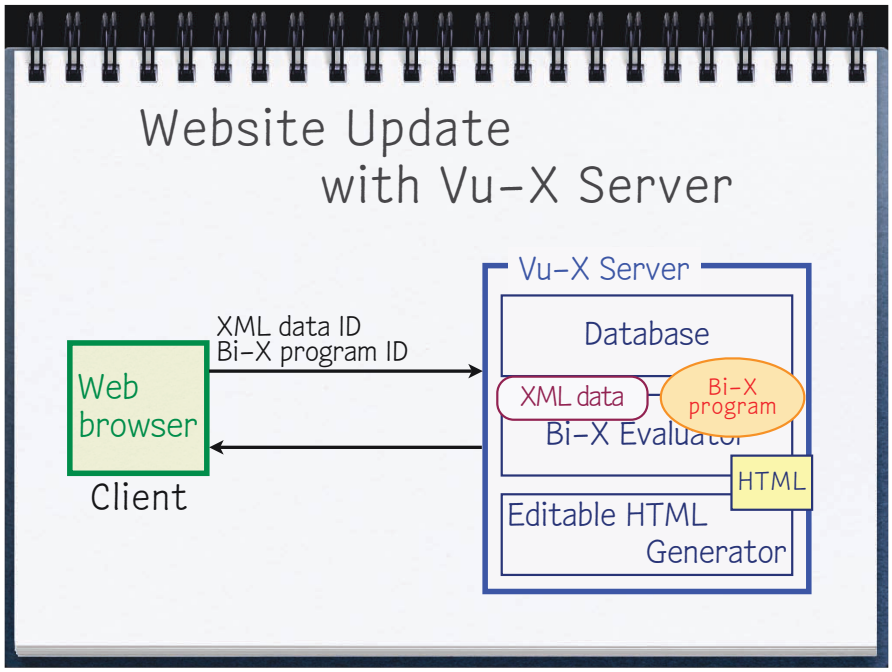
Website Update with Vu-X Server

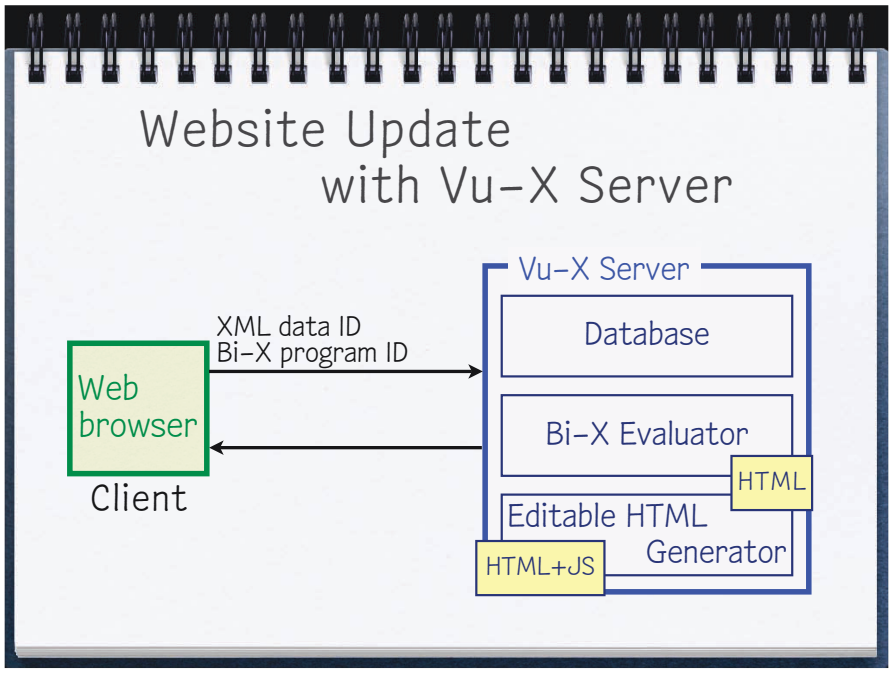


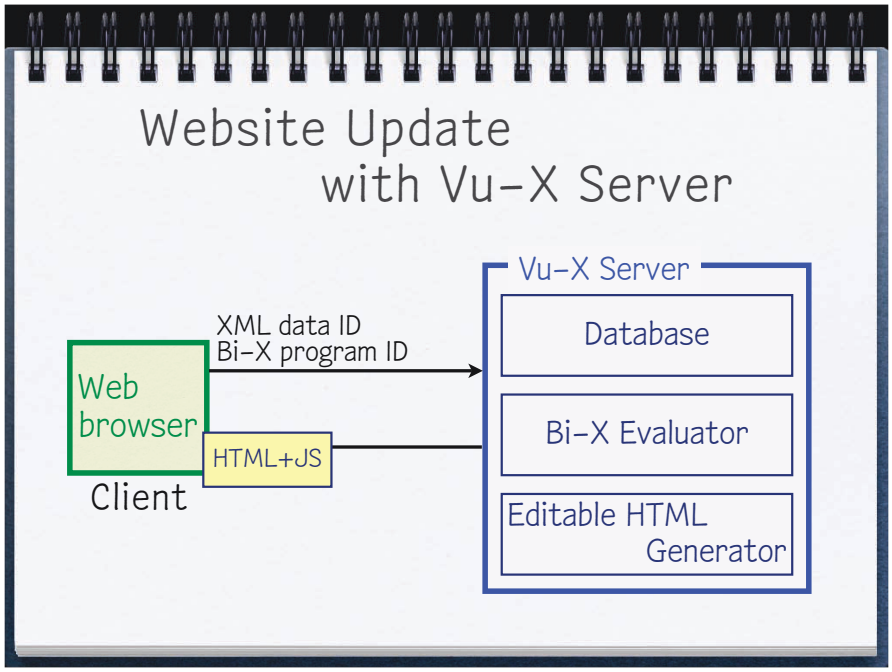
Website Update with Vu-X Server

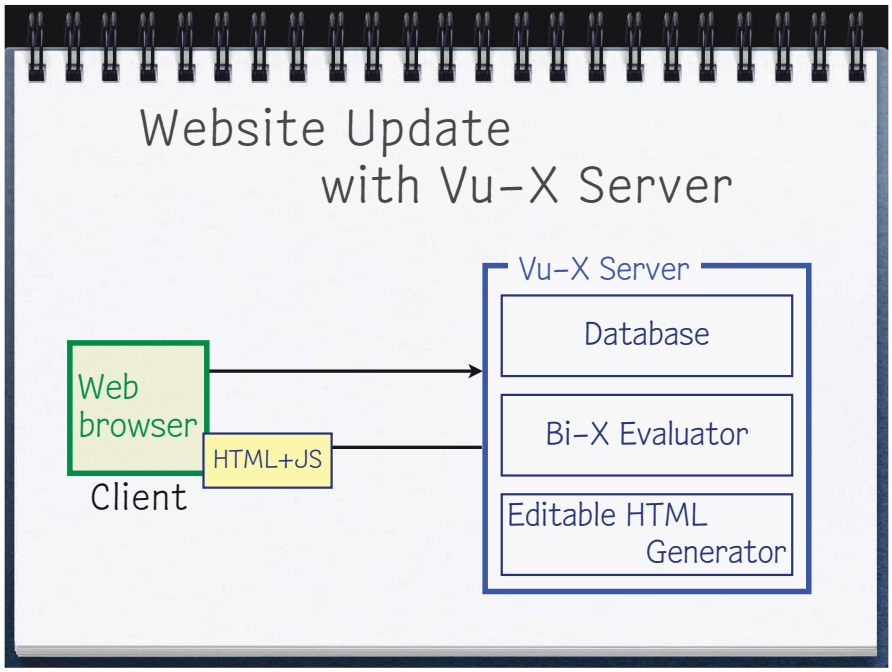


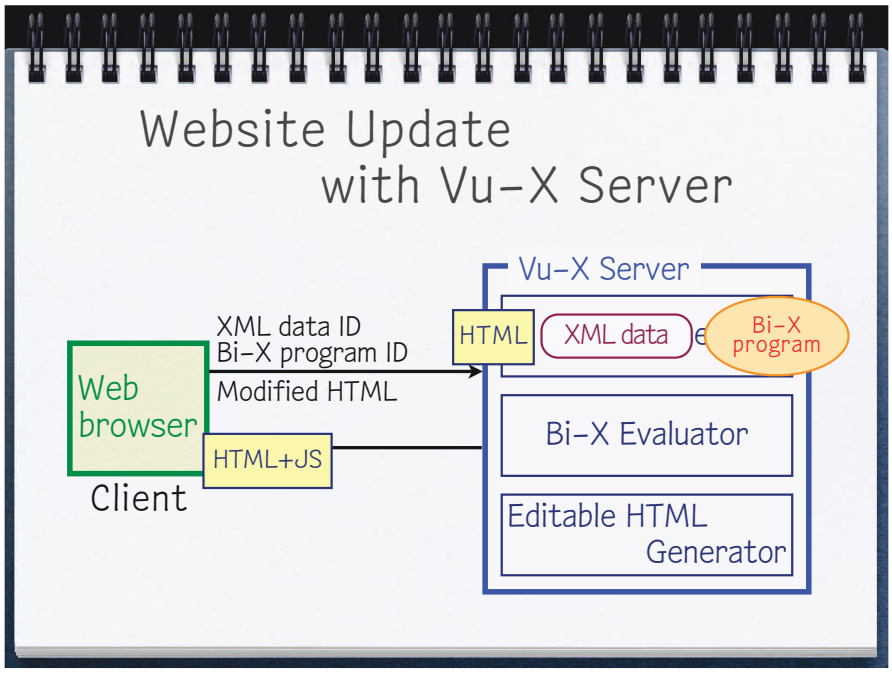

Website Update
with Vu-X Server
Vu-X Server
XML data
Bi-X program
Bi-X Evaluator
Editable HTML Generator
XML data ID
Bi-X program ID
Web browser
Client

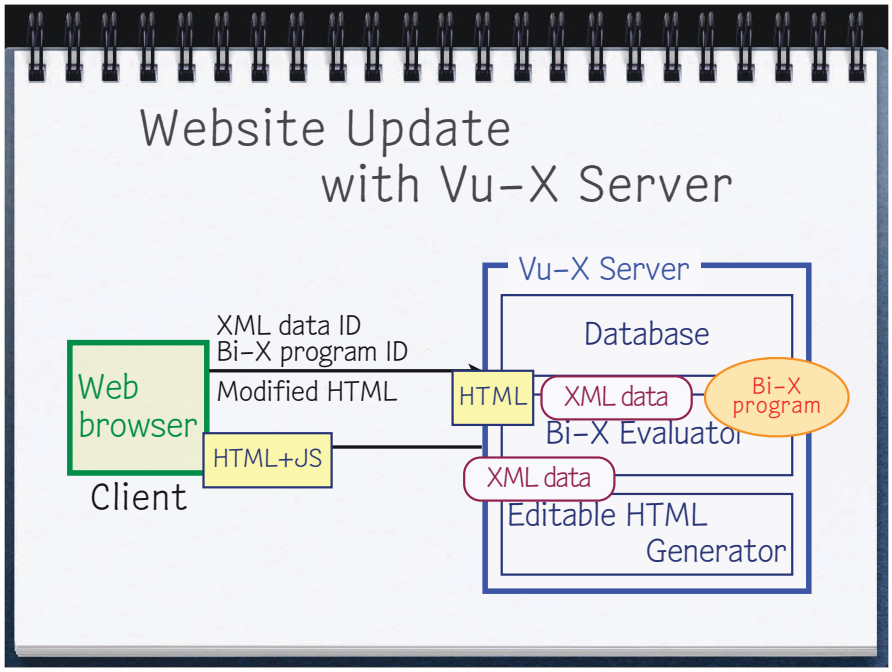

Website Update
with Vu-X Server
Vu-X Server
Database
XML data
Bi-X program
Bi-X Evaluator
HTML
Editable HTML Generator
XML data ID
Bi-X program ID
Web browser
Client

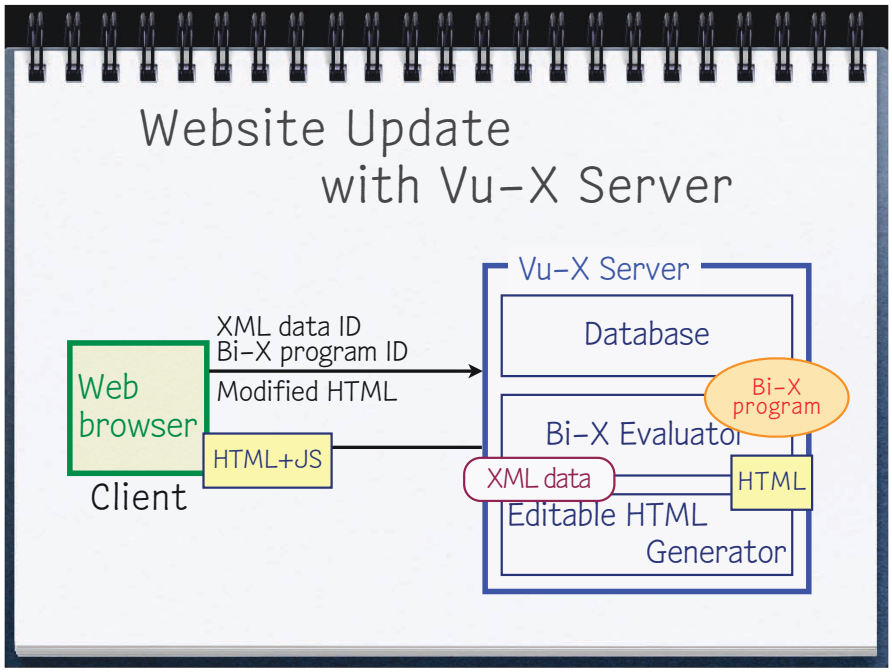

Website Update
with Vu-X Server
Vu-X Server
Database
Bi-X Evaluator
HTML
Editable HTML Generator
HTML+JS
XML data ID
Bi-X program ID
Web browser
Client

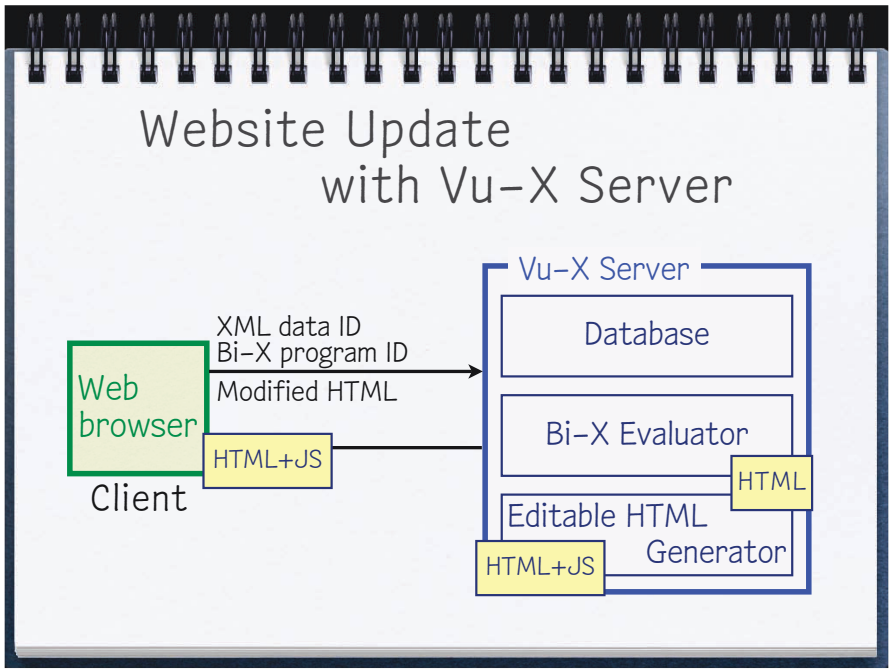

Website Update
with Vu-X Server
Vu-X Server
Database
Bi-X Evaluator
Editable HTML
Generator
XML data ID
Bi-X program ID
Web
browser
HTML+JS
Client


Website Update
with Vu-X Server
Vu-X Server
Database
Bi-X Evaluator
Editable HTML
Generator
Web
browser
HTML+JS
Client

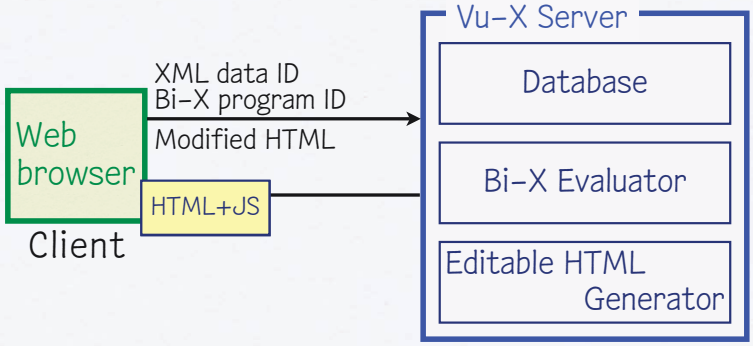

Website Update
with Vu-X Server
Vu-X Server
HTML
XML data
Bi-X
program
Bi-X Evaluator
Editable HTML
Generator
XML data ID
Bi-X program ID
Modified HTML
Web
browser
HTML+JS
Client


Website Update
with Vu-X Server
Vu-X Server
Database
XML data ID
Bi-X program ID
Modified HTML
Web
browser
HTML
XML data
Bi-X
program
Bi-X Evaluator
HTML+JS
XML data
Editable HTML
Generator
Client


Website Update
with Vu-X Server
Vu-X Server
Database
XML data ID
Bi-X program ID
Modified HTML
Web
browser
Bi-X
program
Bi-X Evaluator
HTML+JS
XML data
HTML
Editable HTML
Generator
Client


Website Update
with Vu-X Server
Vu-X Server
Database
XML data ID
Bi-X program ID
Modified HTML
Web
browser
Bi-X Evaluator
HTML+JS
HTML
Editable HTML
Generator
HTML+JS
Client

Website Update with Vu-X Server



Vu-X : Website Updating System based on Bidirectional Transformation

- It is easy to maintain consistency.
  - Vu-X enjoys bidirectional transformation.
- Only one-way transformation is needed.
  - Transformations are written in Bi-X language.
- No special environment is needed.
  - We employ Bi-X server and Vu-X server.
- Each page can be modified on a web browser.
  - We can edit the webpage in WYSIWYG style.

Vu-X Demo

http://www.psdlab.org/vux/

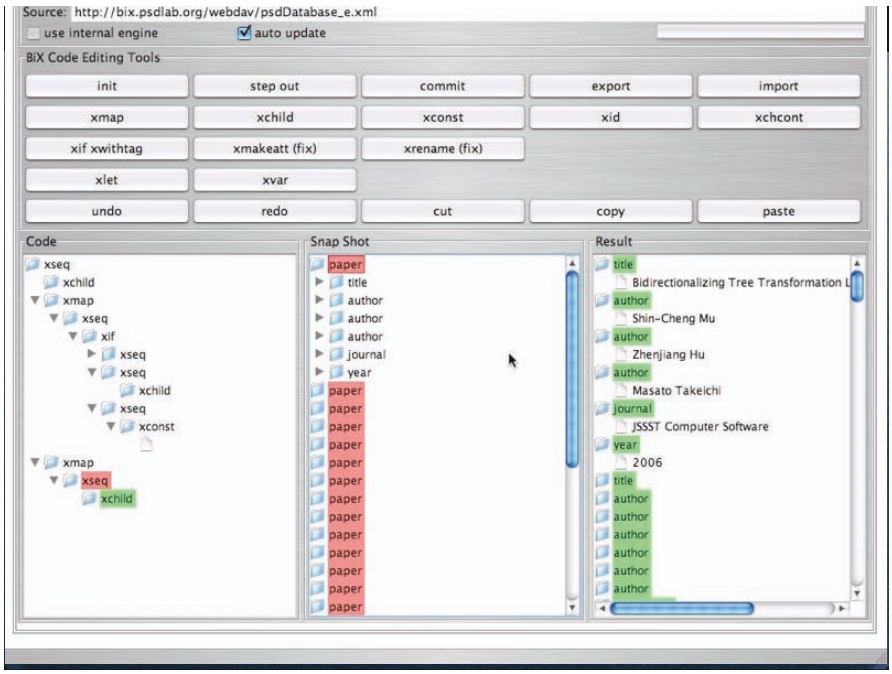
- XML data … psdDatabase_src.xml
- Bi-X code … publication.bix
- Generated HTML … (publication.html)
  - Publication list sorted by year

A problem is remaining...

- We claim that
  - You can easily update your contents without inconsistency from the browsers
  - if you construct your website by XML database and Bi-X program.
- Who writes the Bi-X program?
  - It is complicated because of point-free.

Who writes the Bi-X program?

- Ask Dr. Liu.
  - Is this a solution?
- Use an XQuery-to-Bi-X translation tool.
  - The generated code may be inefficient.
- Use a Bi-X code builder.
  - The program is generated by interactive operation on actual data.



Conclusion

- We presented a website updating system Vu-X (available at http://www.psdlab.org/vux/).
  - It is based on bidirectional transformation.
  - Only forward transformation is required thanks to Bi-X.
  - We can update on a standard web browser in WYSIWYG style.

COMPLEXITY ANALYSIS OF PROGRAMS: METHODS AND CHALLENGES

by Lars Kristiansen

Talk given at 3rd DIKU-IST Joint Workshop on Foundation of Software

5-6 October 2007, Roskilde, Denmark

Some references:

- Kristiansen & Niggl.
On the computational complexity of imperative programming languages.
Theoretical Computer Science 318 (2004).

- Kristiansen & Jones.
The flow of data and the complexity of algorithms.
CiE'05, LNCS 3959 (2005)

- Jones & Kristiansen
A flow calculus of mwp-bounds for complexity analysis.
Submitted.

(1)

The summer of 2000:

I tried to understand Bellantoni-Cook recursion.

I work with simple imperative programming languages. I studied examples like...

```
Y:=1; Z:=1;
loop X { Z:=0
         loop Y { Z:=Z+2 }
         Y:=Z }
```

```
Y:=1; Z:=0;
loop X {
         loop Y { Z:=Z+2 }
         Y:=Y+1 }
```


Observations:

— no "circles" are present inside a loop

⇓

the values computed will be bounded by polynomials in the input (and thus the program will run in polynomial time)

— "circles" can be detected by an algorithm

The paper I wrote with Niggl is based on these observations.

In Kristiansen & Niggl we study a very rudimentary stack language:

push(a, stack) pop(stack) nil(stack)

P;Q if top(stack) = symbol then {P}

foreach S do {P}

We define a measure, i.e. a computable function

$$\nu : \text{programs} \longrightarrow \mathbb{N}$$

Property of ν:

$\nu(P) = 0$ iff no loop of P has a "circle" in its dataflow graph

Theorem.

Ptime =

$\{f \mid \exists P(\nu(P)=0 \text{ and } P \text{ computes } f)\}$

We also characterised the Grzegorczyk class $\mathcal{E}^{i+3}$ as the functions computable by programs of ν-measure $i+1$.

Now, .. why characterise complexity classes ?

$\mathcal{E}^2$ = LINSPACE (Ritchie 1962)

the Cobham class = Ptime (Cobham 1964)

the Bellantoni-Cook class = Ptime (B&C 1992)

There are several motivations, e.g.

- show that the complexity classes are natural mathematical entities not depending on a particular machine model
- understand the complexity classes better, and maybe solve some of the open problems
- understand the computational power of constructions in programming languages, "is iteration more powerful than recursion?" "is LOGSPACE = Ptime?"

(6)

However, in my (& Niggel's) case I felt it was rather pointless to characterise Ptime, and yet we did! (This was the theorem the editor and the referees wanted.)

The most essential parts of the paper was not the characterisation of Ptime ... or LINSPACE ... or the Grzegorczyk classes...

BUT

- the data flow analysis
- the computational method based on this analysis

never explicated;
just burried down in theorems and proofs

(7)

We had programs

P_0 P_1 P_2 P_3 ...

(some running in polynomial time, and some not)

We had a computational method

$\mathcal{M}$: program $\to$ {yes, no}

such that

$\mathcal{M}(P_i) = yes$

$\Downarrow$

P_i runs in polynomial time

I noted two things!

(8)

1. On the one hand, the set

$\{P \mid P \text{ runs in polynomial time}\}$

is undecidable.

2. On the other hand, we had a computional method $\mathcal{M}$ such that

$\{P \mid \mathcal{M}(P) = yes\} \subsetneq \{P \mid P \text{ runs in poly. time}\}$

and there should be possible to find a more powerful method $\mathcal{M}'$ such that

$\{P \mid \mathcal{M}(P) = yes\} \subsetneq$

$\{P \mid \mathcal{M}'(P) = yes\} \subsetneq$

$\{P \mid P \text{ runs in poly time}\}$

Contains more natural programs than

(9)

I could

- extend the language

X := Y
X := X+(Z*Z)
if ? then... else..

- refine the flow analysis

three types of flow:
- m-flow
- w-flow
- p-flow

Terminology:

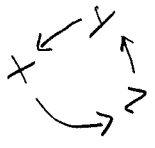
- harmful things are things that cause a program to compute a value not bounded by a polymial in the input
- harmless things are things that are not harmful:

(10)

m-flow:

When m-flow occurs isolated from other types of flow, m-flow is always harmless.

```
Loop U { X := Y;
         Y := Z;
         Z := X }
```

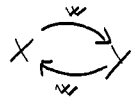


- no need to put restrictions on m-flow
- "circles" may very well occur

(11)

Both w-flow and p-flow might be harmfull. A "circle" of such flow might cause a program to compute a value not bounded by a polynomial in the input.

```
loop Z { X := Y+Y;
         Y := X   }
```

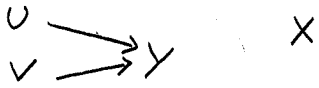


W-flow is iteration-independent

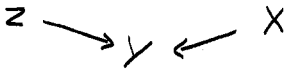
P-flow is iteration-dependent

(12)

```
Loop X { Y := U + V }
```




```
loop X { Y := Y + Z }
```

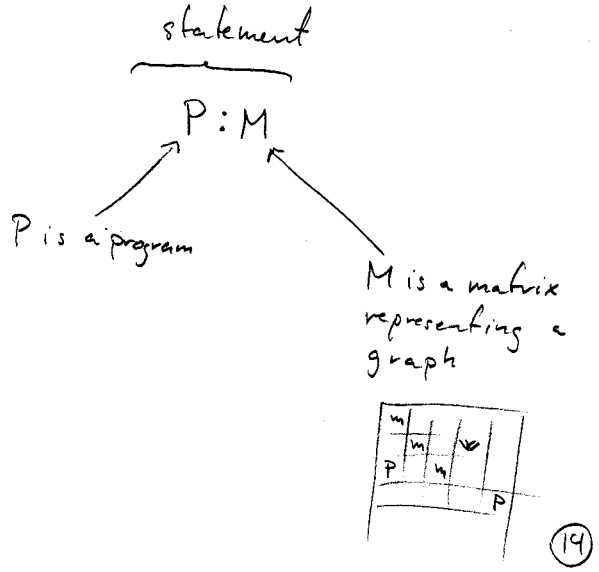


W-flow and p-flow makes it possible to distinguish between the two situations.

(13)

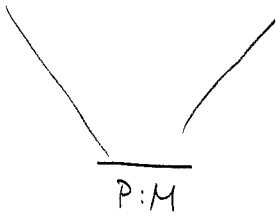
mxp-calculus (joint work with Neil)

We have a syntactic proof calculus for assigning mxp-flow graphs to programs.



(14)

A program P is derivable iff there exists a matrix M and a derivation of the form



Theorem

P is derivable $\Rightarrow$ P runs in polynomial time

This yields a computational method M such that

M(P)=yes $\Rightarrow$ P runs in polynomial time

(15)

How do we argue that a method $\mathcal{M}$ is powerful?

I Empirical research: apply the method to large quanta of programs.

II Argue by (paradigm) examples: "Hey, my method can deal with this well-known sorting algorithm whereas yours cannot."

III Characterise Ptime (and other complexity classes): Prove that for any $f \in$ Ptime there exists a program P such that
- $\mathcal{M}(P) =$ yes
- P computes f

Not a good way to argue!

(16)

IV (my favorite method)

Weaken the semantics such that

P runs in polynomial time under the weak semantics

$\Downarrow\!\Downarrow$

P runs in polynomial time under the standard semantics

Find an interesting subset S of programs.
(The subset S running under the weak semantics cannot yield full Turing computability.)

Prove that

$\{P \mid \mathcal{M}(P) = \text{yes and } P \in S\} =$

$\{P \mid P \text{ runs in poly. time and } P \in S\}$

(17)

That is, prove that the method M
Works perfect for the fragment S
under the weak semantics.

If you succeed, ... fine.

If you fail, ...

you may realise why
you fail,
and then, you may see
how to improve the method.

(18)

We tried to prove that the mwp-method
worked perfect for the language

If ? then ... else ..

for X 1 4

X := expression in *, +, X, Y, Z.

Note:
- no constants
- nondeterministic
  if-then-else

I couldn't do it,
but I believed it was true!

(19)

Amir cooked up the following counter example:

```
loop W
{ if ? then
    { Y := X1 ; Z := X2 }
  else
    { Y := X2 , Z := X1 } ;
  U := Y + Z
  X1 := U }
```

A few weeks later Amir came up with an improved method that probably works perfect for the fragment.

(20)

Challenge:

Extend the current methods to work for richer languages

... to work for real-life programming languages ...

??? Can we do it ???

— The toy languages we have studied is indeed rudimentary, but it is natural and an essential fragment of real-life languages.

— We may possess the key to .....

(21)

Bug Hunting with Coccinelle

Henrik Stuart

Joint work with

Julia Lawall (DIKU)
René Rydhof Hansen (University of Aalborg)

Department of Computer Science, University of Copenhagen

3rd DIKU-IST Joint Workshop on Foundations of Software

Linux memory allocation bugs

Problem:

- Deadlocking the kernel with `kmalloc(e, GFP_KERNEL)` when interrupts are disabled

Goal:

- Use `kmalloc(e, GFP_ATOMIC)` when interrupts are disabled

Linux memory allocation bugfix for `foo()`

```
--- a/foo.c 2007-09-26 21:02:34.000000000 +0200
+++ b/foo.c 2007-09-26 21:02:51.000000000 +0200
@@ -1,7 +1,7 @@
 void foo() {
   cli(); /* disables interrupts */
   bar();
-  buf = kmalloc(size, GFP_KERNEL);
+  buf = kmalloc(size, GFP_ATOMIC);
   baz();
   sti(); /* re-enables interrupts */
 }
```


Linux memory allocation bugfix (SmPL version)

```
@@ expression size; @@

  cli();
  ... WHEN != sti();
- kmalloc(size, GFP_KERNEL)
+ kmalloc(size, GFP_ATOMIC)
```


Why Coccinelle and searching for bugs?

- Good at searching for patterns based on syntax
- Efficient on large code-bases (the Linux kernel)

Use-after-free

```
kfree(data->ptr);


if (foo) {
  warn("Data corruption at %x", data->ptr);
  return E_FAIL;
}
```


Use-after-free

```
kfree(data->ptr);


if (foo) {
  warn("Data corruption at %x", data->ptr);
  return E_FAIL;
}
```

```
@@ expression E; @@
kfree(E)
...
E
```


Use-after-free

```
kfree(data->ptr);

data->ptr = kmalloc(size, GFP_KERNEL);

if (foo) {
  warn("Data corruption at %x", data->ptr);
  return E_FAIL;
}
```

```
@@ expression E; @@
kfree(E)
...
E
```

Use-after-free

```
kfree(data->ptr);

reassign(&data->ptr, some_value);

if (foo) {
  warn("Data corruption at %x", data->ptr);
  return E_FAIL;
}
```

```
@@ expression E; @@
kfree(E)
...
E
```


Buffer overflows

```
int buf[20];
int i;

for (i = 0; i <= 20; ++i)
  buf[i] = i;
```


Buffer overflows

```
int buf[20];
int i;

for (i = 0; i <= 20; ++i)
  buf[i] = i;
```

```
@@ type T; identifier I; constant C; expression E; @@
T I[C];
<... I[E] ...>
```

Buffer overflows

```
int buf[20];
int i;

for (i = 0; i < 20; ++i)
  buf[i] = i;
```

```
@@ type T; identifier I; constant C; expression E; @@
T I[C];
<... I[E] ...>
```


Finding bugs

Problem:

- We can only find things based on its syntax

Finding bugs

Problem:

- We can only find things based on its syntax

Solution:

- Make it possible to use data flow information

Extending Coccinelle

Requirements:

- Easy for experimentation
- Familiar to Linux developers

Extending Coccinelle

Requirements:

- Easy for experimentation
- Familiar to Linux developers

Possible solution:

- Perl

Extending Coccinelle

Requirements:

- Easy for experimentation
- Familiar to Linux developers

Possible solution:

- ~~Perl~~
- Python

Revisiting use-after-free

```
kfree(data->ptr);


if (foo) {
  warn("Data corruption at %x", data->ptr);
  return E_FAIL;
}
```

```
@@ expression E; @@
kfree(E)
...
E
```


Revisiting use-after-free

```
kfree(data->ptr);


if (foo) {
  warn("Data corruption at %x", data->ptr);
  return E_FAIL;
}
```

```
@ rule1 @ expression E, E2; @@
kfree(E)
...
E2
```


Revisiting use-after-free

```
kfree(data->ptr);


if (foo) {
  warn("Data corruption at %x", data->ptr);
  return E_FAIL;
}
```

```
@ rule1 @ expression E, E2; @@
kfree(E)
...
E2
@ script:python @ rule1.E as x, rule1.E2 as y @@
cocci.include_match(x.ast = y.ast and
   cocci.dfa.usedef(x) = cocci.dfa.usedef(y))
```

Revisiting use-after-free

```
kfree(data->ptr);

data->ptr = kmalloc(size, GFP_KERNEL);

if (foo) {
  warn("Data corruption at %x", data->ptr);
  return E_FAIL;
}
```

```
@ rule1 @ expression E, E2; @@
kfree(E)
...
E2
@ script:python @ rule1.E as x, rule1.E2 as y @@
cocci.include_match(x.ast = y.ast and
   cocci.dfa.usedef(x) = cocci.dfa.usedef(y))
```


Finding buffer overflows

```
int buf[20];
int i;

for (i = 0; i <= 20; ++i)
  buf[i] = i;
```

```
@ rule1 @ type T; identifier I; constant C; expression E; @@
T I[C];
<... I[E] ...>
```


Finding buffer overflows

```
int buf[20];
int i;

for (i = 0; i <= 20; ++i)
  buf[i] = i;
```

```
@ rule1 @ type T; identifier I; constant C; expression E; @@
T I[C];
<... I[E] ...>
@ script:python @ rule1.C as x, rule1.E as y @@
buffer_size = cocci.dfa.eval(x)[0]
index_values = cocci.dfa.eval(y)
cocci.include_match(max(index_values) >= buffer_size)
```

Finding buffer overflows

```
int buf[20];
int i;

for (i = 0; i < 20; ++i)
  buf[i] = i;
```

```
@ rule1 @ type T; identifier I; constant C; expression E; @@
T I[C];
<... I[E] ...>
@ script:python @ rule1.C as x, rule1.E as y @@
buffer_size = cocci.dfa.eval(x)[0]
index_values = cocci.dfa.eval(y)
cocci.include_match(max(index_values) >= buffer_size)
```


Future work

- Expressing bug patterns concisely and understandably
- Experimenting with more categories of bugs

Preliminary conclusions

- Use of scripting languages allow easier experimentation
- Coccinelle can actually find bugs
- Several patches derived from its use has been accepted into the Linux kernel

Ranking Functions for Size-Change Termination II
(extended abstract)

Amir M. Ben-Amram and Chin Soon Lee*

Abstract. The Size-Change Termination technique is based on a program abstraction for which termination is decidable. Termination is verified by a set of local termination proofs that account for *all cycles* in a control-flow graph. We present algorithms that construct a *global ranking function* for an SCT instance. Such functions serve as easy-to-check witnesses for termination, and are therefore interesting for purposes of program certification. Their particular form and complexity shed light on the theory of SCT termination proofs.
Our constructions are simpler and more transparent than previously known. They improve the upper bound on the size of the ranking expression from triply exponential to singly exponential. Another contribution is a set of lower-bound results, proving that our constructions are optimal in a certain sense. An interesting point that arises from our constructions is the usefulness of *multisets of data* in ranking expression construction.

1 SCT and Ranking Functions in a Nutshell

Let Val be a well-ordered set of data values. A control-flow graph (CFG) is a directed multigraph (F, C). The nodes are called flow-chart points or just flow-points. The set of arcs from $f \in F$ to $g \in F$ is C_{fg}. One of the nodes, f_0, is *initial* or starting point. All nodes are reachable from f_0. For each $f \in F$, we have a distinct set of *parameters* $Par(f)$, representing data pertinent to describing the program state at this point. For simplicity, all such sets have the same size n. Formally, the set of (abstract) program states is

$$St = \{(f, \sigma) \mid f \in F, \sigma : Par(f) \to Val\}\,.$$

For $f, g \in F$, a size-change graph (SCG) with source f and target g is a bipartite directed graph with source nodes corresponding to $Par(f)$ and target nodes to $Par(g)$. We write this fact as $G : f \to g$. Arcs of G represent constraints on transitions $(f, \sigma) \to (g, \sigma')$. In the ordinary SCT formulation, there are just two types of arcs: a *strict* arc $x \stackrel{\downarrow}{\to} y$ represents strict descent, i.e., $\sigma(x) > \sigma'(y)$. A *non-strict* arc $x \to y$ represents the constraint $\sigma(x) \geq \sigma'(y)$. We write $G \models (f, \sigma) \mapsto (g, \sigma')$ if all constraints are satisfied. An SCT instance, or abstract program, also known as *annotated control-flow graph* (ACG), is a CFG where every arc $c \in C_{fg}$ is annotated with a SCG $G_c : f \to g$.

Let $\mathcal{G}$ be an SCT instance (formally we view $\mathcal{G}$ as just the set of SCG's, implicitly specifying the CFG). A $\mathcal{G}$-*multipath* is a (finite or infinite) sequence $M = G_1 G_2 \ldots$ of elements of $\mathcal{G}$ that label a corresponding directed path in the CFG, often denoted

* benamram.amir@gmail.com, cslee_sg@hotmail.com

by *cs* (for computation sequence, or call sequence—for functional programmers). We also view a multipath as the (finite or infinite) *layered directed graph* obtained by identifying the target nodes of G_i with the source nodes of G_{i+1}. A *thread* in M is a (finite or infinite) directed path in this graph. A thread is *descending* if it includes a strict arc; it is *infinitely descending* if it includes infinitely many strict arcs.

$\mathcal{G}$ is said to *satisfy SCT* (or "terminate") if every infinite multipath contains an infinitely-descending thread. This is a sufficient condition for termination of any program modelled by $\mathcal{G}$ (in fact, the most precise condition). In the rest of this paper, we only consider terminating instances.

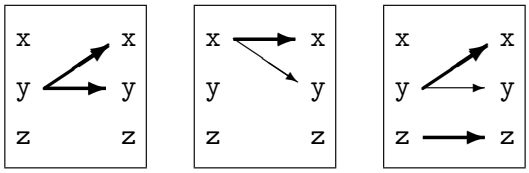
Let $P(s, s')$ be any predicate defined over pairs of states. We write $G \models P(s, s')$ if $G \models s \mapsto s' \Rightarrow P(s, s')$. A (global) ranking function for $\mathcal{G}$ is a function $\rho : St \to W$, where W is a well-ordered set, such that $G \models \rho(s) > \rho(s')$ for every $G \in \mathcal{G}$[1]. It is often convenient to write $\rho(f, [x_1 \to v_1, \ldots, x_n \to v_n])$ as $\rho_f(v_1, \ldots, v_n)$.

Constructing a ranking function for an SCT instance is sometimes a way to understand the type of "behaviour" that the instance expresses. Examples found in previous publications on SCT include programs where the maximum of parameters decreases (consider a standard recursive gcd function), programs where the minimum decreases and programs with a lexicographic descent in a tuple of parameters. It has recently been shown [4] that a ranking function can be constructed for any given SCT instance. It has the following form:

$$\rho(s) = \min(\max S_1, \max S_2, \ldots)$$

where $\max S_i$ represents the maximum element among a set S_i of *vectors* of parameter values and constants, where vectors are lexicographically ordered. We refer to the above form of expressions as $\min[\max[V]]$ where V refers to the type of vectors.

Here is an example: Consider an SCT instance consisting of the graphs $G_1, G_2, G_3 : f_0 \to f_0$ drawn below; the heavy arcs are strict.



A ranking function for $\mathcal{G}$ is $\rho_{f_0}(x, y, z) = \max\{\langle y, 0, z\rangle, \langle x, 1, z\rangle\}$. To verify this, check each graph in turn, considering each possibility among $y > x$, $y = x$ and $y < x$ for the state before and after the transition (plus the constraints expressed in the graph). In this example the *min* operator is unnecessary, so the expression has type $\max[V]$, with $V = Par(f_0) \times \{0, 1\} \times Par(f_0)$.

The above graph set is also an example of a *fan-in free* SCT instance (no two arcs enter the same node). Fan-in free graphs have been identified as an interesting class in previous work [5, 2, 1]. We will also consider the class of fan-out free instances—mostly because they are similar to fan-in free ones but easier to work with.

[1] We may omit explicit references to s, s' for convenience, and use variable names for their values, e.g., writing $G \models x > x'$ instead of the more precise $G \models \sigma_s(x) > \sigma_{s'}(x)$.

2 Statement of Results

Unfortunately, the size limit on this abstract precludes a presentation of our construction techniques. In this section, we state the results with a bit of commentary. The proofs will be published in the full paper.

Definition 2.1 (Tree Expressions). *For a class* `E` *of expressions, a tree expression over* `E` *is either an expression* `e` $\in$ `E` *or a conditional* `if x<y then e1 else e2`*, where* `x`*,* `y` *are parameter names and* `e1`*,* `e2` *are tree expressions over* `E`*.*

Definition 2.2 (Simple Multiset Ordering, SMO). *Let A, B be finite multisets over Val. We write $A > B$ if $|A| > |B|$ (the cardinality of A is larger) or if $|A| = |B|$ and the sets can be listed as $A = \{a_1, a_2, \ldots\}$ and $B = \{b_1, b_2, \ldots\}$ with $a_i \geq b_i$ for all i and $a_i > b_i$ for at least one i. We write $A \geq B$ for the non-strict variant.*

Let $k > 0$. When $|A| = |B| = k$, $A > B$ means that a sorted listing of A is lexicographically greater than a sorted listing of B. This is true for both descending sort and ascending sort; which gives two ways of completing it to a total order over k-element multisets. The first gives the multiset order of [3]; the second, the dual multiset order of [2]. Both are useful in our work. In fact, we need a total order in order to have well-defined min and max operators. We define the min operator to use the dual order, while the max operator is defined by the Dershowitz-Manna order.

Definition 2.3 (vectors). *For $f \in F$ and $B > 0$, V_f^B is the set of vectors $\mathbf{v} = \langle v_1, v_2, \ldots\rangle$, of even length, where for every odd i, v_i is a non-empty subset of $Par(f)$, such that all odd positions constitute a partition of $Par(f)$; while for even i, v_i is an integer between 0 and B. If $B = \omega$, i is any nonnegative integer.*

The *value* of $\mathbf{v} \in V_f$ in a given program state is obtained by substituting values for parameters. The odd positions thus become multisets over *Val*. Such vectors are ordered lexicographically, where multisets are ordered by some extension of SMO, and numbers by the natural order. We use the convention that the value is meant whenever $\mathbf{v}$ is referred to in a context that requires a value, e.g., when making statements about order; the state is supposed to be understood from context.

Theorem 2.4. *Let $\mathcal{G}$ be a terminating SCT instance, with m flow-points and n parameters per point.*

1. *Let $B = (1+m)(n!)(n^2)^{2^n}$. There is a ranking function for $\mathcal{G}$ of the form $\rho_f(\sigma) = \min_{S\in\mathcal{S}_f} \max S$, where $\mathcal{S}_f \subseteq \wp(V_f^B)$, $|S| \leq n!$ for all $S \in \mathcal{S}_f$, and $|\mathcal{S}_f| \leq (B^n n!)^{n!}$. There is also a ranking function where $\rho_f(\sigma)$ is a tree expression over V_f^B at the leaves; the size of the expression is $O(n^n)$.*
2. *If $\mathcal{G}$ is fan-out free, let $B = m \cdot 2^n$. There is a ranking function for $\mathcal{G}$ of the form $\rho_f(\sigma) = \min_{\mathbf{v}\in S_f} \mathbf{v}$ where $S_f \subseteq V_f^B$ and $|S_f| \leq n!$. For fan-in free graphs we have the same result with* max *instead of* min*.*

All the results are given by explicit constructions. In all of them, it is possible to restrict the set components of the vectors to singletons, thus avoiding the use of multiset orders. However, the constructions make use of multisets; we also observe that the use of multisets may help in getting a smaller expression for ρ_f. For a tiny example, consider the size-change graph $\{x \stackrel{\downarrow}{\rightarrow} y, y \rightarrow x\}$; we have the set-valued ranking function $\rho(x, y) = \{x, y\}$. Without multisets, we need a bigger expression.

The size of our ranking functions is a vast improvement over the triply-exponential upper bound of [4]. Is it optimal? Already for the fan-out free case, there is a complexity-theoretic argument against the existence of a polynomially-computable family of ranking functions. More interestingly perhaps, we have explicit constructions that provide tight lower bounds on the size of ranking expressions from a class that generalizes the expressions described in Theorem 2.4.

Definition 2.5. *For a state $s = (f, \sigma)$, let $Order(s)$ be the ordering of the parameter values in s (represented, for example, as a graph on $Par(f)$).*

Definition 2.6. *A VSO function (for Vectors Selected by Order) is a function $\rho_f(s)$ that can be described by assigning to any order τ on $Par(f)$ a vector $\rho^*_f(\tau) \in V^\omega_f$, such that $\rho_f(s)$ is given by evaluating $\rho^*_f(Order(s))$.*

If vectors contain entries that are sets of parameters, one has to specify in which sense they are meant to descend. Our first result refers to SMO, which is the way our ranking functions (Theorem 2.4) work.

Theorem 2.7. *There is a fan-out free, terminating SCT instance $\mathcal{H}$ with a single flow-point f, $n + 1$ parameters and $2n - 1$ size-change graphs, such that any VSO ranking function ρ_f for $\mathcal{H}$ (based on SMO) must use at least $n!$ different vectors.*

The following result concerns the (more flexible) Dual Multiset Order [2].

Theorem 2.8. *There is a fan-out free, terminating SCT instance $\mathcal{K}$ with a single flow-point f, $2n + 1$ parameters and $n + 1$ size-change graphs, such that any VSO ranking function ρ_f for $\mathcal{K}$ (based on DMO) must use at least 2^n different vectors.*

Note that the lower bound dropped from $2^{\Theta(n \log n)}$ to 2^n. It is an intriguing open problem to find out whether such use of multiset ordering can actually improve the upper bound.

References

[1] Amir M. Ben-Amram. Size-change termination with difference constraints. *ACM Transactions on Programming Languages and Systems*, 2007. To appear.

[2] Amir M. Ben-Amram and Chin Soon Lee. Size-change analysis in polynomial time. *ACM Transactions on Programming Languages and Systems*, 29(1), 2007.

[3] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, August 1979.

[4] Chin Soon Lee. ranking functions for size-change termination. Submitted.

[5] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. 28th ACM Symposium on Principles of Programming Languages, pages 81–92. 2001.

Hiding Backtracking Operations in Software Model Checking from the Environment

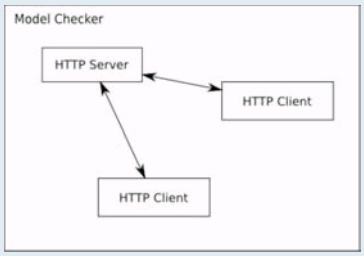
Cyrille Artho, Etsuya Shibayama, Yoshinori Tanabe
National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Masami Hagiya, Watcharin Leungwattanakit
Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo

3rd. DIKU-IST Joint Workshop on Foundations of Software
6 October 2007

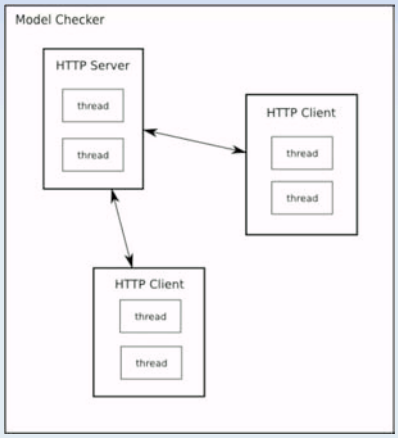
Motivation

- Originally, software model checkers could not handle multi-process networked applications.
- One approach called centralization solved this problem by gathering all peers in a target system and encapsulate them in one process.



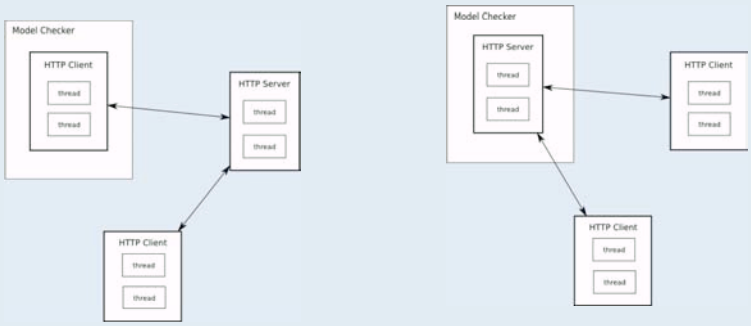
Motivation (cont.)

- All peers must be written in the same programming language.
- The model checker has to simulate interleaving among all processes.
- This centralization approach suffers from state explosion problem.



Objective

- Model check a single peer separately.
- Homogeneous language is not required.
- Scalable with respect to the number of threads.



4

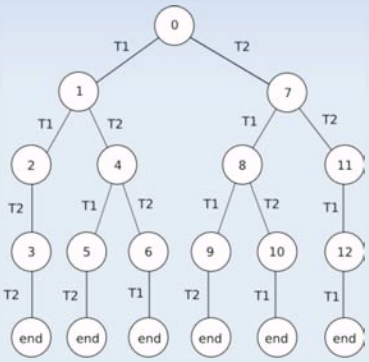
Problem state ment

- Concentrate on applications that communicate with other peers.
- All other peers are outside the model checker.
- Backtracking occurs repeatedly in the model checking process.

5

Problem state ment (cont.)

- Each branch represents a transition sending a message to the server.
- If backtracking occurs, the same message is sent more than once.



6

Solution

- A module that controls communication between the model checked application and its peers.
- Peers are not aware that their counterpart is being model checked.
- The model checker verifies the target application exhaustively.

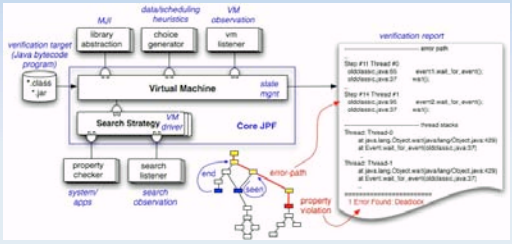
7

Overview

- Introduction to Java PathFinder
- Introduction to Cache layer
- Operation demonstration
- Limitation
- Experiments
- Conclusion

8

Introduction to Java PathFinder

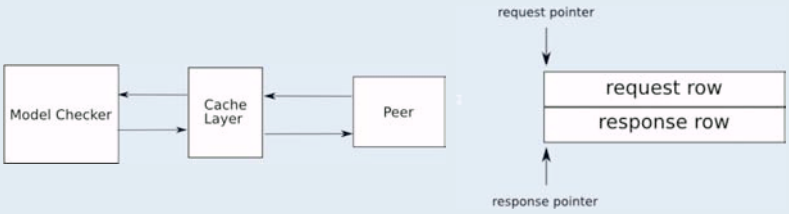


- A model checker for Java bytecode.
- Java Virtual Machine (JVM) inside JPF checks for violation of properties by executing a program in effectively all possible paths of execution.

9

Introduction to Cache Layer

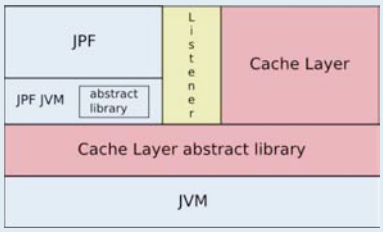
- Cache Layer is placed between the model checked application and the environment.
- Request/Response Record (RRC) is used for storing pairs of messages.



10

Architecture

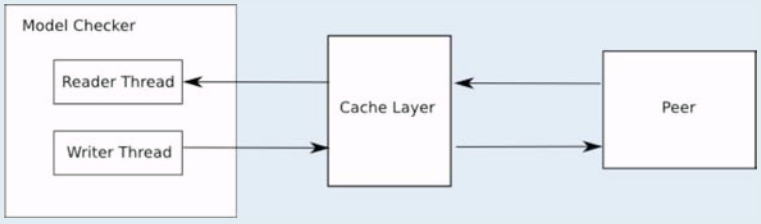
- JPF has its own abstract library.
- The listener links the JPF core and the cache layer.
- The cache layer abstracts some network library classes.



11

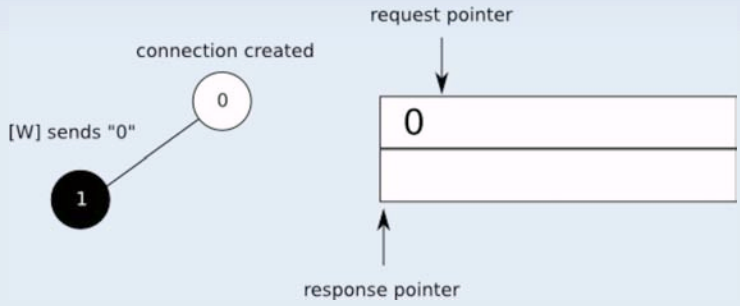
The Cache Layer operation demonstration

- Writer thread sends '0' and '1' to a peer.
- Reader thread reads replied letters from the peer (server).



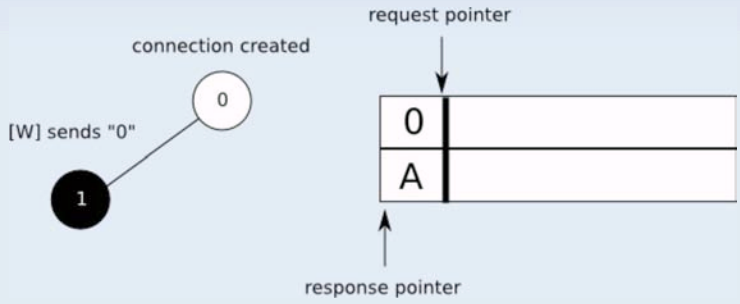
12

The Cache Layer operation demonstration (cont.)



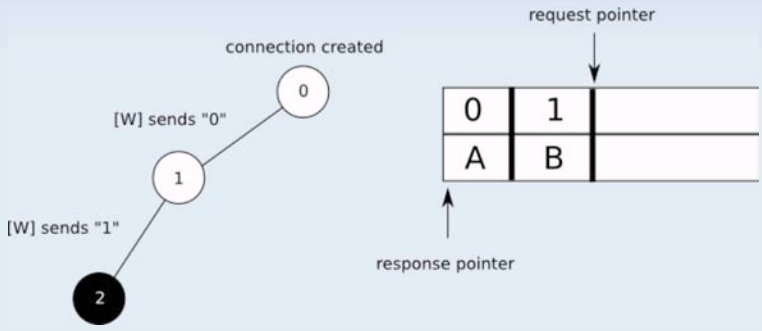
13

The Cache Layer operation demonstration (cont.)



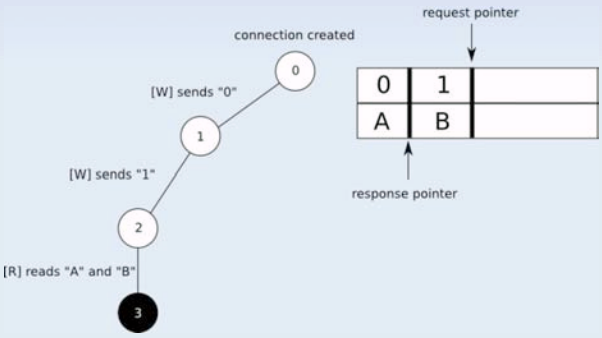
14

The Cache Layer operation demonstration (cont.)



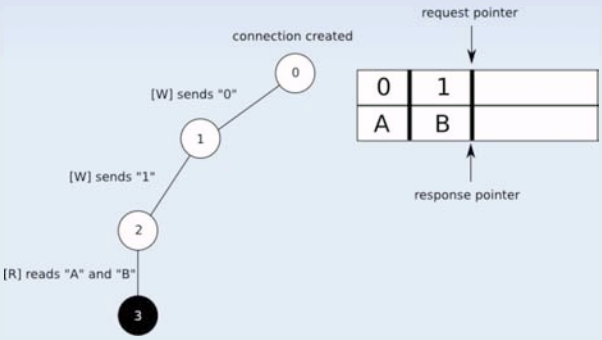
15

The Cache Layer operation demonstration (cont.)



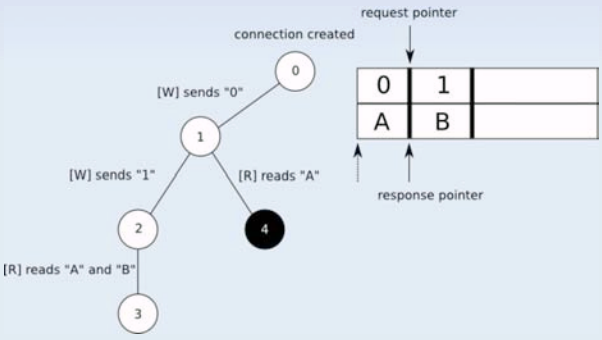
16

The Cache Layer operation demonstration (cont.)



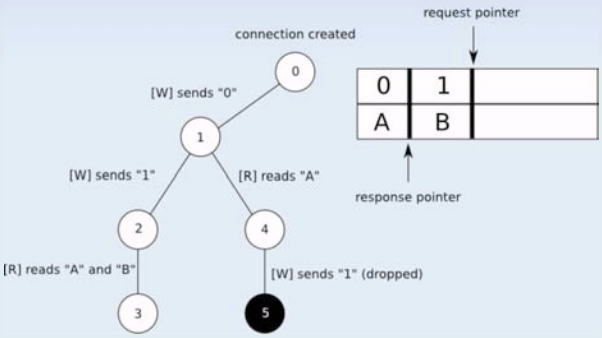
17

The Cache Layer operation demonstration (cont.)



18

The Cache Layer operation demonstration (cont.)



19

Experiments

- Alphabet server/client
- HTTP server/client
- Chat server/client

Configuration	Number of states	Running time (min:sec)
Alphabet client (2 threads, 2 characters)	11386	0:13
Alphabet server (2 threads, 2 characters)	226	0:08
HTTP client (2 threads, 4-byte file)	816	1:07
HTTP server (1 thread, 72-byte file)	13	0:02
Chat client (1 thread, 1 message)	116	0:01
Chat server (1 client, 1 message)	538	0:06

Running on Intel Centrino Duo T2400 with 512 MB memory

20

Limitation

- This approach is limited to systems whose message sequence is independent of thread scheduling.
- The request/response record structure depends on the delay of response messages.

21

Conclusion

- I/O Cache is designed to extend functionality of model checkers for verifying networked applications.
- It plays the opponent-side role of the model-checked application.
- Its mechanism relies on matching between request and response messages transferred over the link.

22

Future work

- Relax the restriction on message sequences.
- Refine the approach to deal with delayed response messages.
- Extend the approach to cover more unreliable protocols such as UDP (User Datagram Protocol).

23

Making operations on standard-library containers strongly exception safe*

Jyrki Katajainen

Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark

Abstract. An operation on an element container is said to provide a strong guarantee of exception safety if, in case an exception is thrown, the operation leaves the container in the state in which it was before the operation. In this paper, we explore how to adjust operations on C++ standard-library containers to provide the strong guarantee of exception safety, instead of the default guarantee, without violating the stringent performance requirements specified in the C++ standard. In particular, we show that every strongly exception-safe operation on dynamic arrays and ordered dictionaries is only a constant factor slower than the corresponding default-guarantee operation. In terms of the amount of space, the overhead introduced is linear in the number of elements stored.

Keywords. C++ standard library, standard template library, data structures, containers, exception safety, commit-or-rollback semantics

1. Introduction

In the C++ community, it has been well-known for a long time that programming with exceptions can be problematic and that the issue of exception safety must be taken seriously (see, for example, [5]). In computer applications, which are supposed to run forever, it is important to catch exceptions whenever these are thrown and to care for proper exception handling. In particular, when an exception is thrown, it must be ensured that the data structures manipulated are not corrupted so that a proper recovery is possible and the execution of an application program can continue after the recovery—without human intervention.

The notion of exception safety is defined as follows (see, e.g. [1, 18]). An *operation* on a container is said to be *exception safe* if the operation leaves the container in a valid state when the operation is terminated by throwing an exception. In addition, the operation should ensure that every resource

*Presented at the "3rd DIKU-IST Joint Workshop on Foundations of Software" held in Roskilde in October 2007.

Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project "Generic programming—algorithms and tools").

CPH STL Report 2007-5, August 2006. Revised October 2007.

that is acquired is (eventually) released. A *valid state* means a state that allows the container to be accessed and destroyed without causing undefined behaviour or an exception to be thrown from a destructor.

A *successful completion* of a container operation means that the operation performs its intended actions correctly without leaking any resources. In C++, operations on standard-library containers are defined to provide the following three kinds of exception safety [1, 18]:

Default guarantee: The operation completes successfully, or throws an exception and maintains the basic invariants of the manipulated container (i.e. keeps the container in a valid state) and leaks no resources.

Strong guarantee: The operation completes successfully (commit), or throws an exception and makes no changes to the manipulated container and leaks no resources (rollback).

No-throw guarantee: The operation always completes successfully and never throws an exception.

All operations on standard-library containers provide the default guarantee, but some key operations also provide the stronger forms of exception safety.

This work is part of the CPH STL project [15], where the goal is to implement an enhanced edition of the STL (Standard Template Library). The CPH STL provides several alternative realizations for various standard-library containers (singly-linked lists, doubly-linked lists, dynamic arrays, double-ended dynamic arrays, ordered dictionaries, unordered dictionaries, priority queues, and double-ended priority queues). In the current form, none of the implementations provide the strong guarantee of exception safety for all operations, but the plan is that in future releases there will exist an implementation of each container class that provides the strong guarantee of exception safety for all operations.

The motivation of writing this paper arose from the project assignments handed out to our graduate students in the last two years in our "Generic programming" course. Students were asked to program a generic library component (priority queue [10] or ordered dictionary [11]) which provides stronger guarantees with respect to runtime performance, space efficiency, iterator validity, and exception safety than those provided by existing realizations available at the Internet. Of the issues considered, exception safety turned out to be the most difficult part of the assignments. Namely, only one group of students, of 30 groups completing their work, succeeded in providing a strongly exception-safe library component. Hence, we must agree with the earlier authors (see, for example, [5, 16]) that it requires extraordinary care to write exception-safe code.

In this paper, advice is given for implementers of the CPH STL, and other generic libraries, how all operations on standard-library containers can be adjusted to provide the strong guarantee of exception safety without violating the performance requirements specified in the C++ standard. The surveys [1, 16, 18, 19], where guidelines for exception-safe programming are given, were the starting point of this research. As summarized in [2] and [18, Table on p. 956], several operations on standard-library containers are only

required to provide the default guarantee of exception safety, or possibly the strong and no-throw guarantees under some specific conditions. Here the aim is to provide the strong guarantee of exception safety efficiently without any conditions.

The rest of this paper is structured as follows. In Section 2, we describe a general technique for converting a program into a form that provides the strong guarantee of exception safety. This technique can be applied in cases where the modifications made by a container operation are easily reversible. In Sections 4 and 5, we consider dynamic arrays and ordered dictionaries, respectively. Our strategy is to show how to implement some decisive operations in an exception-safe manner in the strong sense, and then analyse the efficiency of the proposed implementations. In Section 6, we offer a few concluding remarks. A longer version of this paper is under preparation where we will discuss the issue in greater detail and report experimental results comparing the practical efficiency of various exception-safe implementations of standard-library operations.

2. Techniques for writing exception-safe code

In many of the earlier papers (see, e.g. [16, 18]), guidelines for writing exception-safe code are given. Two of the techniques seem to be more fundamental than others:

T_1: "Never let go a piece of information before its replacement can be stored." When updating an object, one should not destroy its old representation before a new representation is completely constructed and can replace the old version without any risk of exceptions.

T_2: "Resource acquisition is initialization." One can rely on the C++ language rule that when an exception is thrown in the body of a constructor, objects (including bases) already constructed by an initializer will be properly destroyed. (Recall that in constructors objects can be initialized using the initializer list which is a comma-separated list of expressions enclosed in braces.)

Often when a container operation is adjusted to provide the strong guarantee of exception safety, e.g. using the above-mentioned techniques, the program code becomes longer. Next we prove a fundamental theorem which says that this increase cannot be large, provided that the operations performed are easily reversible. In particular, operations that are not reversible introduce the main difficulty when attempting to achieve the strong guarantee of exception safety. For example, element copying is not necessarily reversible since the copy constructor or copy assignment can throw an exception when a rollback of the copy operation is tried.

Given a sequence S of operations, let $\mathit{undo}(S)$ denote the sequence of operations that reverse the effects caused by S, and let $\ell(S)$ denote the length of S measured, for example, in the number of bytes used for its description.

Theorem 1. *Let $k \geq 0$ be an integer. Assume that S_i, $i \in \{1, 2, \ldots, k+1\}$, is a sequence of operations that do not throw any exceptions, and that u_i, $i \in \{1, 2, \ldots, k\}$, is an operation that can throw an exception and is strongly exception safe. Consider now program $P \stackrel{\text{def}}{=} S_1; u_1; S_2; u_2; \ldots S_k; u_k; S_{k+1}$; and let P' be a program that is strongly exception safe and has the same effect as P. Provided that every sequence S_i and operation u_i are easily reversible, e.g. $undo(S_i)$ and $undo(u_i)$ exist, P' can be constructed using the primitives of P and their reversals, and the length of P' is proportional to*

$$\sum_{i=1}^{k} [1 + \ell(u_i) + \ell(undo(u_i))] + \sum_{i=1}^{k+1} [1 + \ell(S_i) + \ell(undo(S_i))] \ .$$

If $\ell(undo(S_i)) = O(\ell(S_i))$ for all $i \in \{1, 2, , \ldots, k+1\}$ and $\ell(undo(u_i)) = O(\ell(u_i))$ for all $i \in \{1, 2, , \ldots, k\}$, the length of P' is linear in the length of P.

Proof. It is straightforward to verify that the following program fulfils the requirements set for program P'.

```
S1;
try {
  u1;
  S2;
  try {
    u2;
    S3;
    ...
  }
  catch(...) {
    undo(S2);
    undo(u1);
    throw;
  }
catch(...) {
  undo(S1);
  throw;
}
```

It is important to note that, since operations u_i are tried, they should not have any unexpected side-effects so it is necessary that they satisfy the strong guarantee of exception safety. □

The proof of the theorem provides a general technique for crafting exception-safe container operations. However, in the form described it can only be applied for straight-line programs. To give the technique wider applicability, we borrow a technique from the database literature:

T_3: "Maintain a log of the structural changes made to facilitate a rollback." It is important that the structural changes made are reversible so that the log can be used to restore the original state of the data structure, if a recovery turns out be necessary.

Now the main concern is how to keep the size of the log reasonable so that the log would not unnecessarily slow down the operation under consideration. Therefore, when technique T_3 is used, some space optimization may be necessary. For example, we keep the log compact by storing there small integers (a few bits) instead of complete full-word integers.

One problem with the techniques discussed is that they often affect the readability of programs and can make maintenance work less attractive. For example, when the resource-acquisition-is-initialization technique (T_2) is used, it may be necessary to create several artificial classes and place the definition of these classes far away from the place where they are used. Similarly, when the technique provided by Theorem 1 is used, there can be a large separation between the code for operation sequence S and that for $undo(S)$. These are the main reasons why the D programming language offers new language constructs to support exception-safe programming (for more details, see [7]).

3. Unsafe and safe operations

All the container classes in the C++ standard library [3] are generic and take several template parameters which make the containers flexible and applicable for many different scenarios. The following set of types may be used for the customization of a container:

- $\mathcal{E}$: the type of the *elements* (or *values*) manipulated;
- $\mathcal{C}$: the type of the *comparator* which is a function object used in element comparisons;
- $\mathcal{A}$: the type of the *allocator* which provides an interface to allocate, construct, destroy, and deallocate objects;
- $\mathcal{N}$: the type of the *compartments* (or *nodes*) used for storing the elements and any related data like pointers to other compartments;
- $\mathcal{I}$: the type of the *iterators* used for referring to the compartments; and
- $\mathcal{S}$: the type of the *data structure* used for storing the compartments.

In the CPH STL, all container classes are bridge classes (for more about the bridge pattern, consult [8]) that take the template parameter $\mathcal{S}$ which can be used to modify the implementation strategy of a container. Often, parameters $\mathcal{N}$, $\mathcal{I}$, and $\mathcal{S}$ are only used for configuration purposes so hereafter we assume that the library, not the user, supplies them.

In general, all user-supplied operations passed via function arguments or template arguments can throw exceptions so these can be problematic when writing exception-safe code. To make the discussion more concrete, consider a binary-heap class that can be used for the realization of a priority queue. According to the declaration [10], the binary-heap class takes four template parameters $\mathcal{E}$, $\mathcal{C}$, $\mathcal{A}$, and $\mathcal{N}$. Of the user-supplied operations, the following five can throw exceptions:

- O_1: copy constructor of an allocator (of type $\mathcal{A}$),

O_2: function `allocate()` of an allocator (indicating that no memory is available),
O_3: copy constructor of an element (of type `E`) (used by function `construct()` of an allocator),
O_4: copy constructor of a comparator (of type `C`), and
O_5: **operator()** of a comparator, or comparator itself if it is a function.

As will be seen, the same set of operations is critical when manipulating the other containers, too.

Basically, all classes with operations that do not throw exceptions are friendly for library implementers. Especially, operations on the following types do not throw exceptions:

- built-in types including pointers,
- types without user-defined operations, and
- functions from the C library (unless they take a function argument that can throw).

In addition, the C++ standard [3] guarantees that no copy constructor or copy assignment of an iterator defined for a standard container throw exceptions.

It is the responsibility of library users to ensure that

- user-defined operations leave container elements in valid states,
- user-defined operations leak no resources, and
- user-defined destructors do not throw exceptions.

Technically, user-defined operations can leave container elements in invalid states and can leak resources, and destructors can throw exceptions, but in normal circumstances the only way to recover from these type of errors is to terminate the program. Especially, operations on the standard-library containers only give their exception-safety guarantees under the assumption that destructors do not throw exceptions.

4. Dynamic arrays

In C++, dynamic (extensible, flexible, or resizable) arrays are called vectors. In the CPH STL, the `vector` class takes three template parameters:

template <
  typename $\mathcal{E}$,
  typename $\mathcal{A}$ = `std::allocator`<$\mathcal{E}$>,
  typename $\mathcal{S}$ = `cphstl::contiguous_vector`<$\mathcal{E}$, $\mathcal{A}$>
>
class `vector`;

The user-supplied operations that can throw exceptions include operations O_1, O_2, and O_3 defined in Section 3. All member functions of the `vector` class that use these three operations must be programmed carefully. In [18], several alternative ways of implementing the constructor, the assignment, `push_back()` in an exception-safe manner are described.

Let us, for example, consider the constructor of the `vector` class. The representation of a `vector` consists of an allocator and a handle to an array of elements. A copy of the allocator (O_1) given as a parameter for the constructor can be created as part of the initializer list (T_2). If an exception is thrown during the copying or later on, the language guarantees that the copy of the allocator is properly destroyed. The potential problems caused by function `construct()` of the allocator (O_2) can be handled by allocating a memory segment for new elements, constructing the elements (O_3), and first after a successful construction updating the handle to point to the appropriate memory segment (T_1). If any of the element constructions fails, the created copies can be easily destroyed since this only involves element destructions. However, in connection with other operations the potential exceptions thrown by the copy constructor of an element (O_3) are more difficult to handle. The main problem is that element copy operations are not necessarily reversible. Therefore, for some operations a standard `vector` provides its stronger-than-default exception-safety guarantees only when element copy operations do not throw exceptions.

If the standard doubling strategy is used to implement a dynamic array, element copying is necessary in connection with each reorganization. A realization based on piecewise-allocated piles, described in [12], is less vulnerable to exceptions thrown during element copying since reorganizations are known not to move elements. In such a realization the only operations that invoke the copy constructor of an element (O_3) in an irreversible manner are `insert()` and `erase()`, which insert elements into or remove elements from the middle of a sequence. In a sense these operations are unnatural for a dynamic array; if these operations were not allowed, a dynamic array could provide the strong guarantee of exception safety without much loss in efficiency.

To get a dynamic array that provides the strong guarantee of exception safety for all the operations specified in the C++ standard—including `insert()` and `erase()`—we have to revert to a less efficient realization. We just use an extra level of indirection: instead of storing elements in an array we store pointers to elements. Now it is easy to insert or remove elements since pointer operations cannot fail and pointer moves are reversible. In case of single-element and multiple-element `insert()`, the construction of new elements is to be tried before any changes are made to the data structure. If element construction fails or if memory allocation fails when reserving additional space for pointers, already constructed elements are destroyed and the temporary array used for storing pointers to elements is released. Single-element and multiple-element `erase()` are even simpler since element destructors are assumed not to throw exceptions and pointer operations are known not to throw exceptions. Copy construction and copy assignment are equally easy since a rollback only involves element destruction and memory disposal.

The above-mentioned implementation strategy relying on an extra level of indirection works for all the current realizations of a dynamic array avail-

able in the CPH STL (see [12, 13]). The main drawback is that element access becomes a constant factor slower. Especially, the usage of pointers may deteriorate the cache behaviour since neighbouring elements are not necessarily stored close to each others in memory.

It should be emphasized that, due to a recent addition made to the C++ standard [3, Clause 23.2.4], neither of the realizations discussed in this section cannot fully replace the standard realization relying on doubling (and halving) since the standard requires that the elements of a `vector` are be to stored contiguously. It seems that with this requirement many `vector` operations cannot be realized in a strongly exception-safe manner with reasonable efficiency.

5. Ordered dictionaries

In C++ terminology, an ordered dictionary is called an *associative container*. Each realization of the C++ standard library should provide four kinds of associative containers: `set` (elements atomic, no duplicates allowed), `map` (elements pairs, no duplicates allowed, ordering based on keys), `multiset` (element atomic, duplicates allowed), and `multimap` (elements pairs, duplicates allowed, ordering based on keys). In the design of the C++ standard library, the main reason to support these four different variants is to allow a direct modification of satellite data, instead of forcing users to invoke `erase()` followed by `insert()` to make such an update. This feature is frequently employed in applications.

An economic way of implementing associative containers is to provide a single search-tree core and then implement `set`, `map`, `multiset`, and `multimap` using the same core—as done in the SGI STL [17]. At the current point, the same strategy is followed in the CPH STL even though in later releases we may go away from this strategy to enhance the efficiency of `multiset` and `multimap` [4]. For the sake of simplicity, let us only consider the `set` class which takes four template parameters:

template <
  typename $\mathcal{E}$,
  typename $\mathcal{C}$ = `std::less`<$\mathcal{E}$>,
  typename $\mathcal{A}$ = `std::allocator`<$\mathcal{E}$>,
  typename $\mathcal{S}$ = `cphstl::red_black_tree`<$\mathcal{E}$, $\mathcal{C}$, $\mathcal{A}$>
>
class `set`;

Now the user-supplied operations that can throw exceptions include operations O_1, O_2, O_3, O_4, and O_5 defined in Section 3.

Of the search trees available at the CPH STL [4, 9, 14], only red-black trees fulfil the strict complexity requirements stated in the C++ standard. Therefore, we will restrict our discussion on them. Since a red-black tree stores the elements in nodes, one element per node, it is relatively easy to make most member functions exception safe. Operations O_1 and O_4 can be

handled using technique T_2 as was done for a dynamic array. Operation O_2 can be handled using technique T_1 by allocating a new node before making any changes to the representation. Similarly, operation O_3 can be tried before making any changes to the old representation. Luckily, the failure in element comparisons (O_5) can also be handled smoothly since the algorithms used for the manipulation of a red-black tree can be partitioned into two phases: a search phase and a rebalancing phase. Element comparisons are only necessary in the search phase and these are done before any changes are made to the representation. All structural changes are done in the rebalancing phase which only involves pointer manipulation and therefore cannot fail.

As pointed out in [18], even if for single-element `insert()` it is easy to provide the strong guarantee of exception safety, for multiple-element `insert()` it is more difficult to provide that guarantee. The reason is that for a naïve implementation based on repeated insertions there is no simple way of reversing the previous successful insertions if an element construction (O_3) or an element comparison (O_5) fails. For red-black trees, in general, one cannot use `erase()` to reverse the effect of `insert()`. As the outcome the container would contain the same elements, but the pointer structure can be much different from that it was before the operations. In many cases for practical purposes such an implementation strategy may be sufficient, but in a strict sense it does not provide the strong guarantee of exception safety.

To provide exception safety in the strong sense, the first thing to do is to perform all node allocations (O_2) and element constructions (O_3) before any updates in the data structure. To keep track of the nodes, a temporary dynamic array is used for storing pointers to the nodes constructed. This can still be seen as an application of technique T_1. The log is maintained for each multiple-element `insert()` separately. As the outcome of the node-construction phase, we have calculated the number of elements being inserted (if the given sequence of elements only provides input iterators, this calculation would not have been possible without any temporary storage). When the number of elements being inserted is known, this can used to allocate space for the log. The space reserved for the log is again freed after completing the multiple-element `insert()` operation under consideration.

Let n denote the number of elements stored in a red-black tree and m the number of elements being inserted as calculated before. Since each insertion on a red-black tree can cause $\Theta(\lg n)$ structural changes, for a naïve implementation the size of the log could be proportional to $m \lg n$. Of course, it is undesirable to have a log of that size. For example, if $m = n$, the size of the log could be $\Theta(n \lg n)$, whereas the data structure itself only uses $\Theta(n)$ space.

To get a more compact representation of the log, we have to look at the structural changes made by a single `insert()` a bit more carefully. There are four types of changes: new nodes are created, nodes are recoloured, left rotations are performed, and right rotations are performed. All these operations are reversible. As observed in [20] (see also [6]), if insertions

are implemented in a bottom-up manner, each insertion performs at most $O(\lg n)$ colour changes followed by at most two rotations. In the log, the ith change can be recorded as a pair (t_i, c_i), where t_i gives the type of the structural change and c_i describes the change itself. If t_i indicates a creation of a new node, c_i can just be a pointer to that node; if t_i indicates a sequence of colour changes, c_i can be the number of levels at which recolourings are necessary (starting from the newly created node); and if t_i indicates a rotation, c_i can be a pointer to the node where the rotation was done. For instance, the reversal of a left rotation is a right rotation at the parent, so the operation is fully reversible when the location of this single node is known. This compaction would reduce the size of the log to $O(m)$ words.

6. Conclusions

We showed how operations on dynamic arrays (C++ standard-library `vector`) and ordered dictionary (C++ standard-library `set`, `multiset`, `map`, and `multimap`) can be adjusted to provide the strong guarantee of exception safety. The techniques used could be applied to all standard-library containers in order to make all operations on them strongly exception safe! Of course, this cannot be achieved without an overhead, but the performance loss caused by the provision of the strong guarantee does not appear to be as high as insinuated in earlier papers. For example, both in [1] and [16], the technique of making a complete copy is offered as an option to achieve the strong guarantee of exception safety. The running time of operations using this approach becomes linear, which is prohibitive for most applications. In sharp contrast our solutions only cause the running time to increase by a constant factor compared to an implementation providing the default guarantee.

The combination of two or more strongly exception-safe operations is not necessarily strongly exception safe (even though the combined operation can be made strongly exception safe using the technique introduced in the proof of Theorem 1). For example, if an element is removed from a dynamic array and inserted into an ordered dictionary, the second operation could fail and the element deleted would be lost even if both operations were strongly exception safe. To simplify the development of exception-safe code, it would be relevant to provide general transaction support in a general-purpose programming language. Such support would be useful in other contexts as well, e.g. when dealing with concurrency. In the programming-language community the issue has been studied under the name "software transactional memory", and there exists systems that can provide transaction support in C++ (see the references mentioned in [21]). It would be interesting to known how efficiently such a general system can provide the strong guarantee of exception safety compared to the direct approach discussed in this paper.

Testing, whether one's code is exception safe or not, is tedious. So far we have done this by visual code inspection. To simplify the testing of error-

handling code, we would need a tool which provides automatic support for reasoning about exception safety. The automated testing framework for verifying exception safety discussed in [1] could be used as a starting point when developing such a tool.

Acknowledgements

I thank Amr Elmasry and Peter Bro Miltersen for discussions that clarified my thoughts on exception safety, and Torben Mogensen for communicating me the idea of nested **try**-**catch** blocks used in the proof of the fundamental theorem on exception safety.

References

[1] D. Abrahams. Exception-safety in generic components: Lessons learned from specifying exception-safety for the C++ standard library. *Selected Papers from the International Seminar on Generic Programming*, *Lecture Notes in Computer Science* **1766**. Springer-Verlag (2000), 69–79.

[2] D. Abrahams and G. Colvin. Making the C++ standard library exception safe. C++ Standards Committee Papers **WG21/N1086R1**. Worldwide Web document available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/` (1997).

[3] British Standards Institute. *The C++ Standard: Incorporating Technical Corrigendum 1*, BS ISO/IEC 14882:2003, 2nd Edition. John Wiley and Sons, Ltd. (2003).

[4] H. Brönnimann and J. Katajainen. Efficiency of various forms of red-black trees. CPH STL Report **2006-2**. Worldwide Web document available at `http://cphstl.dk` (2006).

[5] T. Cargill. Exception handling: A false sense of security. *C++ Report* **6**,9 (1994).

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd Edition. The MIT Press (2001).

[7] Digital Mars. *D programming language*. Website accessible at `http://www.digitalmars.com` (1999–2007).

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley (1995).

[9] J. G. Hansen and A. K. Henriksen. The (multi)?(map|set) of the Copenhagen STL. CPH STL Report **2001-6**. Worldwide Web document available at `http://cphstl.dk` (2001).

[10] J. Katajainen. A meldable, iterator-valid priority queue. CPH STL Report **2005-1**. Worldwide Web document available at `http://cphstl.dk` (2005–2006).

[11] J. Katajainen. Project proposal: Associative containers with strong guarantees CPH STL Report **2007-4**. Worldwide Web document available at `http://cphstl.dk` (2007).

[12] J. Katajainen and B. B. Mortensen. Experiences with the design and implementation of space-efficient deques. *Proceedings of the 5th Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**. Springer-Verlag (2001), 39–50.

[13] M. D. Kristensen. Vector implementation for the CPH STL. CPH STL Report **2004-2**. Worldwide Web document available at `http://cphstl.dk` (2004).

[14] S. Lynge. Implementing the AVL-trees for the CPH STL. CPH STL Report **2004-1**. Worldwide Web document available at `http://cphstl.dk` (2004).

[15] Performance Engineering Laboratory, University of Copenhagen. *The CPH STL*. Website accessible at `http://cphstl.dk` (2000–2007).

[16] J. W. Reeves. Using exceptions effectively: Part I—Coping with exceptions. Worldwide Web document available at `http://www.bleading-edge.com/Publications/C++Report/v9603/Article2a.htm` (1998).

[17] Silicon Graphics, Inc. *Standard template library programmer's guide*. Website accessible at `http://www.sgi.com/tech/stl` (1993–2006).

[18] B. Stroustrup. *Appendix E: Standard-library exception safety. The C++ Programming Language*, Special Edition. Addison-Wesley (2000).

[19] B. Stroustrup. Exception safety: Concepts and techniques. *Advances in Exception Handling Techniques*, *Lecture Notes in Computer Science* **2022**. Springer-Verlag (2001), 60–76.

[20] R. E. Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Information Processing Letters* **16** (1983), 253–257.

[21] Wikipedia. Software transactional memory. Worldwide Web document available at `http://en.wikipedia.org` (2007).

Author Index

After Work
Sunset at Roskilde Fjord