TR-CS-98-09

International Refinement Workshop
&
Formal Methods Pacific ’98

Work-in-progress papers of IRW/FMP’98
29 September – 2 October 1998, Canberra, Australia

Jim Grundy, Martin Schwenke and
Trevor Vickers (editors)

September 1998

Joint Computer Science Technical Report Series
Department of Computer Science
Faculty of Engineering and Information Technology
Computer Sciences Laboratory
Research School of Information Sciences and Engineering
This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports  
Department of Computer Science  
Faculty of Engineering and Information Technology  
The Australian National University  
Canberra ACT 0200  
Australia

or send email to:

Technical.Reports@cs.anu.edu.au

A list of technical reports, including some abstracts and copies of some full reports may be found at:

http://cs.anu.edu.au/techreports/

Recent reports in this series:

TR-CS-98-08 Jim Grundy and Malcolm Newey (editors).  

TR-CS-98-07 Peter Strazdins.  

TR-CS-98-06 M. Manzur Murshed.  

TR-CS-98-05 M. Manzur Murshed.  
Optimal computation of the contour of maximal elements on constrained reconfigurable meshes. May 1998.


Preface

The International Refinement Workshop and Formal Methods Pacific 1998 (IRW/FMP'98), as the title suggests, is a combined event. FMP'98 itself incorporates the 7th Australasian Refinement Workshop and the 4th New Zealand Formal Program Development Colloquium, and follows the inaugural FMP'97 in Wellington.

The Australasian Refinement Workshop (ARW) series began in 1990, to provide a forum for researchers in program refinement. The New Zealand Formal Program Development Colloquium (NZFPDC) series began in 1994 with similar aims. In 1997 the ARW and NZFPDC joined forces in Wellington under the umbrella name Formal Methods Pacific (FMP). The more general name, and regional focus better reflects the broader range of interests of participants and presented papers, while satisfying the need for a regional conference covering formal methods.

The event became the International Refinement Workshop with the postponement of the 7th British Refinement Workshop, and the welcome support of IRW/FMP'98 by the British Computer Society FACS.

IRW/FMP'98 is being held concurrently with The 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98) at The Australian National University.

Papers for IRW/FMP'98 were sought in two streams: Completed Work and Work in Progress. The Work in Progress stream provides a forum for discussion of work that has not yet reached maturity. Submissions were vetted for relevance, but not refereed. This volume presents the Work in Progress papers.

We wish to thank our sponsors for their support: the Department of Computer Science at The Australian National University, and the Australian Research Council (through their Special Research Initiatives Program). The latter grant (sought in conjunction with TPHOLs'98) also enabled us to offer travel bursaries to students to assist their attendance at the conference. Finally, the Cooperative Research Centre for Advanced Computational Systems assisted with the production costs of this volume.

Jim Grundy
Martin Schwenke
Trevor Vickers

Canberra, September 1998
# Table of Contents

Reasoning about Grover’s Quantum Search Algorithm using Probabilistic up .................................................. 1  
*Michael Butler and Pieter Hartel*

Care2 and its Type System ..................................................... 13  
*Keith Harwood*

A Proof Framework with IO Regular Expressions for Dataflow Networks .................................................. 22  
*Yoji Hayashi, Tadashi Yamaguchi and Eiichi Miyamoto*

Can Component Object Model (COM) be Formalized? - The Formal Semantics of the COMEL Language ................... 40  
*Rosziati Ibrahim and Clemens Szyperski*

Modelling Specification-based Testing of Interactive Systems .................................................. 58  
*Ian MacColl and David Carrington*

A Superposition Refinement of Component-based Systems .................................................. 70  
*winnie Weiqun Qiu and John Zic*

I Went Down To The Crossroads: Conjoining Catamorphisms .................................................. 85  
*Martin Schwenke*

Using the Object Calculus to build an Object-Oriented Refinement Calculus .................................................. 97  
*Jamie Shield and David Carrington*

τ-Equivalences and Refinement .................................................. 110  
*Igor V. Tarasyuk*

Local Stores in C++ .................................................. 129  
*Mark Utting and James Dunwoody*

Teaching Formal Methods Lite .................................................. 145  
*Mark Utting and Steve Reeves*

Issues for a Temporal Refinement Calculus .................................................. 159  
*Ron van der Meyden and Yoram Moses*

Author Index .................................................. 167
Reasoning about Grover’s Quantum Search Algorithm using Probabilistic \textit{wp}

Michael Butler and Pieter Hartel

Dept. of Electronics & Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ
United Kingdom
mjb,phh@ecs.soton.ac.uk.

\textbf{Abstract.} Grover’s search algorithm is designed to be executed on a quantum mechanical computer. In this paper, the probabilistic \textit{wp}-calculus is used to model and reason about Grover’s algorithm. It is demonstrated that the calculus provides a rigorous programming notation for modelling this and other quantum algorithms and that it also provides a systematic framework of analysing such algorithms.

1 Introduction

Quantum computers are a proposed means of using quantum mechanical effects to achieve efficient computation. Quantum mechanical systems may be in superpositions of several different states simultaneously. The central idea of quantum computers is to perform operations on these superposed states simultaneously and thus achieve a form of parallel computation. These devices were proposed in the early 1980’s [1,4].

One essential phenomenon of quantum mechanics is that the measurement of a superposed system forces it into a single classical state. Each superposed state is present with a certain amplitude and an observation causes it to collapse to the that state with a probability that depends on its amplitude. This means that, although many computations may be performed in parallel on a quantum device, the result of only one of these may be observed. This may seem like a severe limitation, but several ingenious algorithms have been devised which work by increasing the amplitude of the desired outcome before any observation is performed and thus increasing the likelihood of the observed outcome being the desired one.

One such algorithm is Grover’s quantum search algorithm [7] which performs a search on an unstructured search space of size \( N \) in \( \mathcal{O}(\sqrt{N}) \) steps. To find the desired search value with 100% probability in such a space, a classical computer cannot do better than a linear time search. Grover’s algorithm performs operations on a superposition of all possible search values that serve to increase the amplitude of the desired search value. Grover shows that within \( \mathcal{O}(\sqrt{N}) \) steps there is a greater than 50% chance of finding the desired search value. Boyer et
al [3] proved a stronger result for the algorithm showing that the correct search value can be found in \( O(\sqrt{N}) \) with almost 100% probability.

In this paper, we apply the probabilistic weakest-precondition (wp) calculus of Morgan et al [9] to Grover’s algorithm to redevelop the result of Boyer et al in a more systematic way. The probabilistic \( wp \)-calculus is an extension of Dijkstra’s standard \( wp \)-calculus [5] developed for reasoning about the correctness of imperative programs. The extension supports reasoning about programs containing probabilistic choice. The measurement of a quantum superposition is an example of a probabilistic choice.

Use of the probabilistic \( wp \)-calculus contributes two essential ingredients to the analysis of quantum algorithms. Firstly it provides an elegant and rigorous programming language for describing quantum algorithms. The existing literature uses block diagrams and structured English which can be cumbersome and potentially ambiguous. Secondly, the probabilistic \( wp \)-calculus provides a set of rules for the systematic analysis of the correctness of algorithms. In the case of standard algorithms, the calculus is used to determine whether a program achieves some desired outcome. In the case of probabilistic algorithms, the calculus is used to reason about the probability of a program achieving some desired outcome.

This paper is not simply about re-presenting a known result about Grover’s algorithm but it also aims to demonstrate that the probabilistic \( wp \)-calculus is suitable for both modelling and reasoning about a quantum algorithm. Boyer et al have already derived the same result that we derive here but they do so in a less systematic way. Our hope is that the approach used here could be applied fruitfully to other quantum algorithms and may even aid the development of new quantum algorithms.

The paper is organised as follows. In Section 2, we give a sufficient overview of quantum theory. In Section 3, we present our approach to modelling quantum computation using the programming language of the probabilistic \( wp \)-calculus. In Section 5, we give a sufficient introduction to the rules of the probabilistic \( wp \)-calculus and in Section 6, we use the \( wp \)-calculus to derive a formula for the probability of success of Grover’s algorithm.

## 2 Quantum Systems and Qubits

In quantum mechanics, a superposition of two states \( \alpha \) and \( \beta \) is represented in Dirac’s notation as follows:

\[
S = \alpha |\alpha\rangle + \beta |\beta\rangle.
\]

System \( S \) is said to be in a superposition of \( \alpha \) and \( \beta \). \( |\alpha\rangle \) and \( |\beta\rangle \) are the \textit{basis states} and \( \alpha \) and \( \beta \) are amplitudes. The amplitudes may be complex numbers.

Let \( |z|^2 \) be the square norm\(^1\) of complex number \( z \). Observation of \( S \) will cause the system to collapse to state \( |\alpha\rangle \) with probability \( |\alpha|^2 \) and to \( |\beta\rangle \) with

\(^1\) The square norm of any complex number \( a + bj \) is \( a^2 + b^2 \).
probability $|\beta|^2$. The probabilities must sum to 1:

$$|\alpha|^2 + |\beta|^2 = 1.$$ 

A qubit is a two state quantum system in which the basis states are labelled 0 and 1:

$$S = \alpha|0\rangle + \beta|1\rangle.$$ 

A classical bit has $\alpha = 1$ and $\beta = 0$ or $\alpha = 0$ and $\beta = 1$.

A qubit evolves from one superposition to another using a quantum gate (or function) $F$:

$$F(\alpha|0\rangle + \beta|1\rangle) = \alpha'|0\rangle + \beta'|1\rangle.$$ 

$F$ must be unitary which means that

- probabilities are preserved: $|\alpha'|^2 + |\beta'|^2 = |\alpha|^2 + |\beta|^2$, and
- $F$ has an inverse.

In quantum mechanics, a transformation $F$ is usually modelled using matrix multiplication:

$$F(\alpha|0\rangle + \beta|1\rangle) = U_F \times \begin{pmatrix} \alpha \\ \beta \end{pmatrix},$$

where $U_F$ is a $2 \times 2$ unitary matrix. Matrix $U$ is unitary if $UU^\dagger = U^\dagger U = I$ where $U^\dagger$ is the conjugate transpose of $U$.

A quantum superposition may have an arbitrary number of basis states, not just two. An $N$-state superposition is represented as:

$$S = \sum_{i=0}^{N-1} \alpha_i |i\rangle.$$ 

Observation of $S$ will cause it to collapse to state $|i\rangle$ with probability $|\alpha_i|^2$. Again, the probabilities must sum to 1:

$$\sum_{i=0}^{N-1} |\alpha_i|^2 = 1.$$ 

A quantum register is a collection of qubits and an $L$-qubit register gives rise to a system with $2^L$ basis states. Like qubits, quantum registers evolve under unitary transformations.

For further details on quantum computation, the reader is referred to papers such as [2, 6].
3 Modelling Quantum Computers

A quantum computer is a collection of quantum registers and quantum gates. In this section, we introduce ways of modelling various aspects of quantum computation using the programming language of the probabilistic $\lambda_p$-calculus. We use a subset of the language which includes standard assignment, probabilistic assignment, sequential composition and simple loops.

Firstly, we model an $N$-state quantum system as a function from state indices to complex numbers: $S : (0..N - 1) \to \mathbb{C}$.

A superposition of the form

$$\sum_{i=0}^{N-1} \alpha_i |i\rangle$$

is modelled by the function $S$ where for $0 \leq i < N$:

$$S(i) = \alpha_i.$$

A classical state $i$ is modelled by the function which is zero everywhere except at $i$ which we write as $|i\rangle$:

$$|i\rangle(j) = 1, \text{ if } i = j$$

$$= 0, \text{ otherwise.}$$

Transformation of a quantum state is modelled by a standard assignment statement:

$$S := F(S).$$

$F$ must be unitary for this to be a valid quantum transformation.

We shall find it convenient to use lambda abstraction to represent transformations: $(\lambda i \mid 0 \leq i < N \cdot E)$ represents the function that takes an argument $i$ in the range $0..N - 1$ and returns the value $E$. For example, the unitary transformation that inverts the amplitude of each basis state is modelled as follows:

$$S := (\lambda i \mid 0 \leq i < N \cdot -S(i)).$$

Sequencing of transformations is modelled using sequential composition: let $T_1$ and $T_2$ be transformations, then their sequential composition is written $T_1; T_2$.

The loop which iterates $C$ times over a transformation $T$ is written $\text{do } C \text{ times } T \text{ od}$.

We model the observation of a quantum system using a probabilistic assignment statement. In the simple case, this is a statement of the form:

$$x := E @ p,$$

$$F @ (1 - p).$$
This says that $x$ takes the value $E$ with probability $p$ and the value $F$ with probability $1 - p$. For example, a coin flip is modelled by

$$
\text{coin} := \text{head} @ 0.5,
$$
$$
\text{tail} @ 0.5.
$$

Observation of a two state superposition forces the system into a classical state. This is modelled with the following probabilistic assignment:

$$
S := |0\rangle @ |S(0)|^2,
$$
$$
|1\rangle @ |S(1)|^2.
$$

A generalised probabilistic statement has the form

$$
x := E_i @ p_i, \quad 0 \leq i < N,
$$

where $\left(\sum_{i=0}^{N-1} p_i\right) = 1$.

Now observation of an $N$-state quantum system $S$ may be modelled by

$$
S := |i\rangle @ \|S(i)\|^2, \quad 0 \leq i < N.
$$

That is, $S$ collapses to the classical state $i$ with probability $|S(i)|^2$.

4 Grover’s Search Algorithm

The Grover search problem may be stated as follows:

Given a function $f : (0..N-1) \rightarrow \{0, 1\}$ that is zero everywhere except for one argument $x_0$, where $f(x_0) = 1$, find that argument $x_0$.

The algorithm makes use of the mean of a superposition $S$, written $\overline{S}$, where

$$
\overline{S} = \frac{\sum_{i=0}^{N-1} S(i)}{N}.
$$

The algorithm is represented in the programming language of the probabilistic $wp$-calculus in Figure 1. The initialisation of this algorithm sets the system $S$ up in an equal superposition of all possible basis states. Successive iterations of the loop then serve to increase the amplitude of the search argument $x_0$ while decreasing the amplitude of the other arguments. To see why this is so, consider the case of $N = 8$. The initialisation sets $S$ up in an equal superposition of the eight possible states, represented diagrammatically as follows:

```
  0 1 2 3 4 5 6 7
```
\[ S := (x_i \mid 0 \leq i < N \cdot \frac{1}{\sqrt{N}}) ; \quad \text{Init} \]
\[
\text{do } C \text{ times}
\]
\[
S := (x_i \mid 0 \leq i < N \cdot S(i) - 2f(i) . S(i) ) ;
\]
\[
S := (x_i \mid 0 \leq i < N \cdot 2S - S(i) )
\]
\[
\text{od ;}
\]
\[
S := |i| \otimes |S(i)|^2, \quad 0 \leq i < N \quad \text{Measure}
\]

The first step of the loop body replaces each \( S(i) \) with \( S(i) - 2f(i) . S(i) \). This inverts \( S(i) \) about the origin in the case that \( f(i) = 1 \) and leaves \( S(i) \) unchanged in the case that \( f(i) = 0 \). Assuming that \( f(4) = 1 \), this replaces our example superposition with

```
- - - - - - - - - - - - - - - - - - - - - Average
```

The second step of the loop body inverts each amplitude about the average of all the amplitudes resulting in:

```
- - - - - - - - - - - - - - - - - - - - -
```

The amplitude of state \( |4\rangle \) has increased as a result of the two steps of the loop body, while the amplitude of the others has decreased.

After an optimum number of iterations, \( C \), the amplitude of \( |x_0\rangle \) approaches 1 while the amplitude of the other states approaches 0. An observation is then performed. Since the amplitude of \( |x_0\rangle \) approaches 1, the probability of the observation yielding \( |x_0\rangle \) is close to 1. \( C \) depends on the number of states \( N \) and, as discussed in the next section, it is \( O(\sqrt{N}) \).

5 Probabilistic \textit{wp}

In two-valued logic, a \textit{predicate} may be modelled as a function from some state space to the set \{0, 1\}. For example, the predicate \( x > y \) evaluates to 1 in a state in which \( x \) is greater then \( y \) and evaluates to 0 in any other state. A \textit{probabilistic predicate} generalises the range to the continuous space between 0 and 1 [9]. For
example, the probabilistic predicate $0.5 \times (x > y)$ evaluates to 0.5 in a state in which $x$ is greater than $y$ and evaluates to 0 in any other state.

In the standard $wp$-calculus, the semantics of imperative programs is given using weakest-precondition formulae: for program $prog$ and postcondition $post$, $wp(prog, post)$ represents the weakest precondition (or maximal set of initial states) from which $prog$ is guaranteed to terminate and result in a state satisfying $post$.

The $wp$ rule for assignment is given by:

$$wp( x := E, \ post ) = post[x/E].$$

Here, $post[x/E]$ represents predicate $post$ with all free occurrences of $x$ replaced by $E$. For example,

$$wp(x := 7, \ x > y) = x > y \ [x/7]$$
$$= 7 > y.$$  

That is, the assignment $x := 7$ is guaranteed to establish $x > y$ provided $7 > y$ initially.

The $wp$ rule for sequential composition is given by:

$$wp(\ prog1; prog2, \ post ) = wp(\ prog1, \ wp(prog2, \ post ) ).$$

Both of these rules also apply in the probabilistic $wp$-calculus. The $wp$ rule for simple probabilistic assignment [9] is given by:

$$wp( x := E @ p, \ F @ 1 - p , \ post ) =$$
$$p \times post[x/E] + (1 - p) \times post[x/F].$$

In the case of non-probabilistic $post$, $wp(prog, post)$ represents the probability that program $prog$ establishes $post$. For example

$$wp( coin := head @ 0.5, \ tail @ 0.5, \ coin = head )$$
$$= \text{by (3)}$$
$$0.5 \times (coin = head [coin/head]) + 0.5 \times (coin = head [coin/tail])$$
$$= \text{substitution}$$
$$0.5 \times (head = head) + 0.5 \times (tail = head)$$
$$= 0.5 \times 1 + 0.5 \times 0$$
$$= 0.5.$$

That is, a coin flip establishes $coin = head$ with probability 0.5.

The $wp$ rule for the generalised probabilistic assignment is given by:

$$wp( x := E_i @ p_i, \ 0 \leq i < N, \ post ) = \sum_{i=0}^{N-1} p_i \times post[x/E_i].$$
The only other programming construct we need in order to model Grover's algorithm is the DO-loop. Since the algorithm only loops a constant and finite number of times, we can model \texttt{do C times prog od} as a finite sequential composition of \(C\) copies of \(prog\) which we write as \(prog^C\). We have that

\begin{align*}
prog^0 &= \text{skip} \\
prog^{i+1} &= \text{prog} \; \text{;} \; \text{prog}^i.
\end{align*}

Here, \(\text{skip}\) is the statement that does nothing, with \(wp(\text{skip}, \text{post}) = \text{post}\). The semantics of more general looping constructs is given by least fixed points in the usual way, but we do not need that here.

6 Reasoning about Grover

The postcondition we are interested in for the Grover algorithm is that the correct solution is found, i.e., \(S = |x_0\rangle\). The probability that Grover establishes \(S = |x_0\rangle\) is given by \(wp(\text{Grover, } S = |x_0\rangle)\), so we shall calculate this.

The Grover algorithm has the following structure:

\begin{align*}
\text{Init} ; \\
\text{do C times} \\
\quad \text{Body} \\
\text{od} ; \\
\text{Measure}.
\end{align*}

which we shorten to

\(\text{Init} ; \; \text{Body}^C ; \; \text{Measure}\).

When calculating a formula of the form \(wp(\text{prog1} ; \text{prog2}, \text{post})\), we first calculate \(wp(\text{prog2}, \text{post})\) and then apply \(wp(\text{prog1}, \text{post})\) to the result of this. Thus, to calculate \(wp(\text{Grover, } S = |x_0\rangle)\), we first calculate \(wp(\text{Measure, } S = |x_0\rangle)\):

\begin{align*}
wp(\text{Measure, } S = |x_0\rangle) &= wp(\; S := |i\rangle @ \|S(i)\|^2, \; S = |x_0\rangle) \\
&= \text{by (4)} \\
&= \sum_{i=0}^{N-1} \|S(i)\|^2 \times (|i\rangle = |x_0\rangle) \\
&= \text{since } (|i\rangle = |x_0\rangle) \text{ is 0 for } i \neq x_0 \\
&= \|S(x_0)\|^2.
\end{align*}

Next we try to calculate \(wp(\text{Body}^C, \|S(x_0)\|^2)\). \(\text{Body}^C\) is defined recursively by (5) and (6) so we shall develop recursive equations for \(wp(\text{Body}^C, \|S(x_0)\|^2)\). First we look at the weakest precondition of a single iteration. Let \(P[S]\) stand for a predicate \(P\) containing one or more free occurrences of variable \(S\) and \(P[S']\)
stand for $P$ with all free occurrences of $S$ replaced by $S'$. It is easy to show, using (1) and (2), that

$$wp(\text{Body}, P[S]) = P[S']$$

where $S'(i) = 2\overline{S} - \frac{4}{N} S(x_0) + (2f(i) - 1)S(i)$. 

From (8), we have that

$$wp(\text{Body}, |S(x_0)|^2 ) = |S'(x_0)|^2$$

$$= |2\overline{S} - \frac{4}{N} S(x_0) + (2f(x_0) - 1)S(x_0)|^2$$

$$= |2\overline{S} + (1 - \frac{4}{N})S(x_0)|^2.$$ 

Now this has the form $\|A\overline{S} + BS(x_0)\|^2$ and using (8) we can again show that for any values $A,B$:

$$wp(\text{Body}, |A\overline{S} + BS(x_0)|^2 ) = |A'\overline{S} + BS(x_0)|^2$$


This recurring structure suggests that we define $A_i$ and $B_i$ as follows:

$$A_{i+1} = A_i + 2B_i$$

$$B_{i+1} = \frac{N.B_i - 2.A_i - 4.B_i}{N},$$

to give

$$wp(\text{Body}, |A_i\overline{S} + B_iS(x_0)|^2 ) = |A_{i+1}\overline{S} + B_{i+1}S(x_0)|^2. \quad (10)$$

By induction over $j$, we get

$$wp(\text{Body}^j, |A_i\overline{S} + B_iS(x_0)|^2 ) = |A_{i+j}\overline{S} + B_{i+j}S(x_0)|^2. \quad (11)$$

We take $A_0 = 0$ and $B_0 = 1$ and apply $\text{Body}^C$ to (7) as follows:

$$wp(\text{Body}^C, |S(x_0)|^2 )$$

= since $A_0 = 0, B_0 = 1$

$$wp(\text{Body}^C, |A_0\overline{S} + B_0S(x_0)|^2 )$$

= by (11)

$$\|A_{C}\overline{S} + B_{C}S(x_0)\|^2.$$

Finally, we apply the initialisation to this:

$$wp(\text{Init}, |A_{C}\overline{S} + B_{C}S(x_0)|^2 )$$

$$= \|A_{C}\overline{S} + B_{C}S(x_0)|^2$$

$$= \|A_{C} + B_{C}|^2$$

$$= \frac{1}{\sqrt{N}}$$

$$= \frac{1}{\sqrt{N}}$$

$$= \frac{1}{\sqrt{N}}.$$
Thus we have shown that:

$$wp(\text{Grover}, \quad S = |x_0\rangle) = \left\| \frac{A_C + B_C}{N} \right\|^2.$$  

That is, the probability, $P(C, N)$, of observing the correct value after $C$ iterations is:

$$P(C, N) = \left\| \frac{A_C + B_C}{N} \right\|^2.$$  

Now using standard techniques for finding closed forms for recurrence equations, we can derive the following closed form for $P(C, N)$:

$$P(C, N) = \sin^2((2C + 1)\theta_N)$$

where $\theta_N = \arcsin \frac{1}{\sqrt{N}}$.

This is the same as the formula presented in [3].

![Graph showing probability $P(C)$ for Grover search with $N = 128$.](image)

**Fig. 2.** Probability for Grover search with $N = 128$.

It is interesting to note that $P(C, N)$ is periodic in $C$. This can be seen clearly in Figure 2 which graphs $P(C, N)$ against $C$ for $N = 128$. Here, an optimum probability of success is reached after eight iterations, where $P(8,128) = 0.996$. After eight iterations, the probability starts to decrease again. The reason for the decrease is that, after eight iterations, the average immediately after the inversion about the origin operation goes below zero.

We wish to determine the optimum number of iterations for a given $N$. $P$ reaches a maximum (and a minimum) for a given $N$ when:

$$\frac{d}{dx} P(x, N) = 0.$$  

It is easy to show that the first maximum for a given $N$ is reached at

$$\frac{\pi}{4\theta_N} - \frac{1}{2} \quad \text{where} \quad \theta_N = \arcsin \frac{1}{\sqrt{N}}.$$
We call this $H(N)$. Thus the number of iterations in the Grover algorithm for a search space of size $N$ should be the closest whole number to $H(N)$. In Figure 3, we graph $H(N)$ and indicate that it is $O(\sqrt{N})$.

![Graph showing $\sqrt{N}$ and $H(N)$ vs. size of search space](image)

**Fig. 3.** Optimum number of iterations versus search space size.

7 Conclusions

We have shown how Grover’s search algorithm may be represented in the programming notation of the probabilistic $wp$-calculus. Any quantum computation consists of unitary transformations and probabilistic measurement and these can be modelled in this notation. Thus any quantum algorithm may be modelled in the notation. We believe that this language provides a more rigorous and elegant means of describing quantum algorithms than is normally used in the literature.

We have also shown how the rules of the probabilistic $wp$-calculus may be used to derive a recursive formula for the probability that Grover’s algorithm finds the required solution. Using standard techniques, as described, for example, by Knuth [8], we were able to then find a closed form for this probability which corresponds to the formula presented in [3]. The $wp$-calculus provides a clear and systematic means of stating the required outcome and of deriving the probability of achieving it. Of course, it does not provide everything for free as we still had to use intelligence in recognising the recurring structure and in finding a closed form.

In the case of Grover, we were able to derive an exact probability for success because the algorithm iterates a fixed number of times. Some algorithms iterate until some condition is met rather a fixed number of times. One such example is a generalisation of Grover’s presented in [3] which deals with the situation where there are an unknown number of values $x$ satisfying $f(x) = 1$. In a case like this, we need to find the expected number of iterations rather than the probability of success. For future work, we intend to look at how these cases may be reasoned about using the probabilistic $wp$-calculus.
Acknowledgements

Thanks to Tony Hey and other members of the Quantum Sticky Bun Club for inspiration and to Peter Hoyer for comments on a draft of the paper.

References

Care2 and its Type System

Keith Harwood
Vital Mission Software Pty. Ltd.
485 Windsor Road
Baulkham Hills
Australia, 2153

Abstract. A simple type system is the foundation of an extensive mathematical tool-kit for software engineers to use in applying formal methods in an industrial setting.

1 Care2

Care stands for Computer Assisted Refinement Engineering. It is an attempt to make formal methods available in industry. Care began in 1979, inspired by Eric Hehner[4] and the observation that transforming a recursive form of an algorithm into a loop form was a mechanical task routinely performed by compilers. It developed over the years and was first exposed in public in [2]. Between 1992 and 1995 it was a cooperative venture between Telecommunications Pty. Ltd. and the Software Verification Research Centre of the University of Queensland, in whose hands is its continuing development. The latest paper on Care is [5] and many others are available as technical reports from the SVRC. For a recent bibliography see their Web site at http://svrc.it.uq.edu.au/pages/CARE.html.

Care2 is a start-again-from-scratch attempt to build on the strengths of the original Care and to alleviate some of its shortcomings.

The chief shortcoming of Care is that it looks like a programming language. Even worse, it behaves like a programming language. The user has a worksheet which contains, among other things, specifications for fragments of the code being created. The user selects refinements for the specifications from a library and the tools apply those refinements to produce code and, typically, further specifications. However there is nothing to prevent the user from creating refinements ‘by hand’ and, in fact, the tools occasionally compel the user to do just that. What’s more, one worksheet generates one file in the target programming language, which only enhances the perception that you can ‘write a program in Care’. Even worse, those of us working one the project fell into this trap many times. For example, in spite of strenuous effort to avoid it, [3] still reads like the description of a language.

In Care2, however, there is no worksheet. Instead, all the fragments for a project (which may include several related products and many software components) are maintained in one database and the packaging of those fragments into target language files is made late in the development and can vary from release
2 The Type System

CARE2’s type system is very simple. It has no built-in types, no type constructors and only one relation between types. To create a type you give it a name and a description. The description should explain the rôle that the type will play in the problem at hand. When the customer says, ‘what’s in it for me’, this is where you look. It should not describe either the mathematical properties of the type nor its implementation; those are handled elsewhere. The name should be a mnemonic for the description. Both the name and description are intended for the human reader, they have no significance to the system.

Types in CARE2 are purely syntactic. They are solely concerned with what is or is not a well-formed formula.

Each formula exists in some context. The type system determines which formulæ are acceptable in that context. In use, to enter a formula you must select a context, the system will display the formulæ which are acceptable in that context and you select one of those. Thus, the system enforces type correctness at the time formulæ are created and there is no subsequent type check needed. It is not possible to enter a formula which is type-incorrect. When an operator takes a particular type at some argument position, that argument provides a context in which any function that yields a result of that type is acceptable, so only those functions are displayed for you to select.

No, that’s not quite true. CARE2 allows you to assert that one type is a subtype of another. If we assert that type $S$ is a subtype of type $T$ (written $S < T$) then in any context that type $T$ is acceptable, so is type $S$. Simple though it is, this subtype relation allows us to express many important theorems as a few subtype relations usually with a theorem asserting that the values of one subtype are distinct from those of another. For example, by defining a non-zero subtype for a numerical type and providing a theorem that the values of that type are distinct from zero, the theorem ‘thou shalt not divide by zero’ can be expressed by stating that the division function takes the non-zero type as its second argument. That way, you can’t enter an expression in which the denominator might be zero into the system, nor will the simplification routines or the theorem prover ever create such an expression. The theorem itself never appears explicitly.
In other cases the use of subtypes make theorems simpler by removing pre-
conditions. If \( L \) is a type of list and NEL is a type of non-empty \( L \), rather than write
\[
\forall x : L \ . \ \# x > 0 \Rightarrow \langle \text{head}(x) \rangle \sim \text{tail}(x) = x
\]
we simply write
\[
\forall x : \text{NEL} \ . \ \langle \text{head}(x) \rangle \sim \text{tail}(x) = x
\]
This gives us a unit clause for the prover and, since it is an equality, a candidate
for a re-write rule.

The principal can be applied ad-nauseum. Any predicate expression with one
unbound variable is a candidate for the creation of a subtype. If type \( Bbh \) is a
list of bibliographic references and \( E[x] \) is an expression with \( x \) unbound and of
type \( Bbl \) asserting that \( x \) is sorted then you can create a subtype \( SortedBbl \) with
\( SortedBbl < Bbl \) and
\[
\forall x : SortedBbl \ . \ E[x]
\]
If the expression occurs in more than a couple of places it is worthwhile doing
this as a matter of course. What’s more, if the predicate expression consists of
a number of conjunctions it’s worthwhile creating a subtype for each conjunc
t and making the one you really want a subtype of all of those. Why is it worth-
while? Because in developing code to create such objects there’s a good chance
you’ll have to establish the conjuncts one at a time anyway. This way the type
system keeps track of them rather than developing a longer and longer string
of preconditions all of which get applied in one fell swoop when you reach your
goal.

The original \texttt{CARE} included a powerful algebraic simplifier and a Not Terribly
Bright automatic theorem prover. \texttt{CARE2} will have the same. The type system
assists these and makes them more efficient by preventing useless inferences (if
an inference is type-incorrect there is no theorem that can discharge its proof).
In [7] my colleague Trudy Wei\ss\penalty0 bel shows that a resolution prover restricted in
this way is both correct (which is pretty obvious) and complete (which isn’t).

But to do all that means we have to know things about the values of the
types and I said above that the type system was purely syntactic. And what use
is a type system that is purely syntactic anyway? We need to be able to say that
such-and-such a value is of type so-and-so and therefore it has certain properties.
To do this, the semantic information about a type is embodied in the operators
which specify values of the type and the theorems which describe relationships
between those values. So, if you want a type to represent natural numbers you
need to invent a function to give you a zero value of that type and another
function to give you the successor to a value and a theorem about the successor
function being injective and it’d be nice if you also could do induction ... and
so on. In other words a type gives natural numbers only because the functions,
predicates and theorems that apply to that type generate natural numbers.

But, you don’t want to have to invent the theory of natural numbers from
scratch every time you need one, do you? The good news is that you don’t have
to. The even better news is that you’re not allowed to.
3 Scenarios

When you first create a type you can’t do very much with it. To go further you need to introduce operators and theorems that relate to values of that type. The first step is to associate the type with a scenario.

A scenario is a template can be instantiated to provide the operators and theorems of a family of related types. One scenario provides theories for integers and natural numbers. Another provides theories for simple containers such as sets, lists and trees. Other scenarios provide other theories which are useful in writing software.

However, when you associate a type with a scenario (you instantiate the scenario with the type), it doesn’t give the type any new properties. All it does is inform the system that when you do give the type some properties they will be drawn from that scenario. Further, when you instantiate a scenario with two different types the types are not in any way related. If you have a type \textit{Nlelems} which you want to use for counting the elements of a container type you must instantiate it using the Number scenario, but that doesn’t make it \textit{Nlelems} a number. If you have another type \textit{EventCnt} which you want to use for counting the number of events detected by your system you will also use the Number scenario, but that doesn’t make it a number either, nor does it make it the same type as \textit{Nlelems}.

They only become numbers when you draw material from the scenario. To initialise variables you are going to need a zero, so you ask the scenario to introduce the zero function into your project. Later, you may need addition, so you ask for that function. Later still, you may need to compare two numbers so you ask for the less-than predicate and perhaps a theorem about ordering on that predicate. It is this material that makes the type a number. If it looks like a number and acts like a number, it’s a number. But you have to do this separately for both \textit{Nlelems} and \textit{EventCnt}, and the functions and predicates you get for the two types are different. They might look the same — using anything but ‘0’ to represent the zero would do more harm than good — but the system knows which is which and if you select an \textit{Nlelems} context you will be presented with the \textit{Nlelems} zero and in a \textit{EventCnt} context you’ll get that type’s zero. (The symbol used to represent a function is a property of the function and the user can change it at any time. Naturally, any such change affects every occurrence of the symbol where-ever it occurs. However, the underlying function remains unchanged.)

The reason for this rigmarole is threefold. Firstly, it allows us to ensure that we never use a value intended for one use for something else. This is familiar to users of strong type systems, such as in Ada, where the accepted wisdom is to invent special types for each usage; you can do whatever you want on purpose, but you make it hard to do anything by accident. Here we are applying the same idea to the specification of programs as well as to their implementation.

The second reason is to reduce the size of the theories that the system has to deal with. You introduce a particular function only when you want to describe a particular value, you introduce a predicate only when you need to describe a
particular relationship between values and you introduce a theorem only when it is needed to justify some particular refinement. So the theory that you use is very rarely complete. Why do this? If you introduce more theory than you need you may very well find yourself trying to prove things that are irrelevant to the problem at hand. For example, in Z sequences are modelled as functions whose domain is a segment of the natural numbers starting from one. If you have the full theory on hand you may find yourself having to prove that for some sequence this is in fact the domain, when you know perfectly well that it doesn’t matter, that only the order of the elements matters. When you are doing refinement by hand you can simply ignore this requirement, but CARE2 is supposed to be fully formal, and there is no way to ignore some proof obligation once it has been introduced. On the other hand, if you have never brought this bit into the theory, you can never have to prove it.

Or to put it in other terms, never put off until tomorrow something that you can put off until the day after. Which leads to the third reason for the rigmarole. You don’t have to decide what a type is to be right away. You start with a type that’s just a type. Then you associate it with a scenario. If you are going to count things and do arithmetic of some kind you associate it with the Number scenario. This will give you natural numbers and integers, but you don’t have to make up your mind which you want. If in the course of development you need some property that only natural numbers can supply then when you bring that property into your development the type becomes a type of natural numbers, If you need a property that only integers can supply the type becomes an integer type. (If you need both, you are in trouble and need to back-pedal a bit. Perhaps you really needed two different types rather than one. On the other hand, if you need neither you have put off the decision not till the day after, but forever.) The distinction between types within a scenario is called a variant. The variation between types instantiated from the same scenario depends on whether or not some crucial theorem holds.

In the case of the Number scenario the theorem is that zero is the smallest number. If this holds for a particular type, that type is a natural number; if it doesn’t, it is integer. If it doesn’t matter whether it holds or not then it doesn’t matter whether the type is a natural number of an integer.

You don’t have to bring the crucial theorem into your development, however. The scenarios are designed so that if anything which depends on the theorem is brought in the type becomes the required variant. If you have a number type and want to introduce negation, you will get the integer variant. On the other hand, if you introduce the theorem ‘zero is a left identity for the \textit{maz} function’ you’ll get the natural number variant.

The table below shows the variants of the Container scenario. It depends on three theorems, whether the join function\footnote{The join function, as the name suggests, joins two containers; it is usually called concatenation for lists and union for sets.} is associative, commutative or idempotent. Of the eight possibilities, five seem to be useful.
EmptyT < T
\emptyset : EmptyT
NonemptyT < T
SingletonT < NonemptyT
\text{mk} : \text{Elem} \rightarrow \text{SingletonT}
ProperT < NonemptyT
\text{join} : \text{NonemptyT} \times \text{NonemptyT} \rightarrow \text{ProperT}

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Mobile</th>
<th>List</th>
<th>Bag</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assoc</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Commut</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Idempot</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\forall x : \text{SingletonT}, y : \text{ProperT}. x \neq y \implies \text{NonemptyT} < \text{ProperT}

Table 1. Variants of the Container Scenario

As the table shows, the scenarios are profligate in creating new types. T is
the base type, but there are subtypes for empty containers, non-empty contain-
ers, containers with one member and, apparently, containers with more than
one member. The proper type is the result of the join-proper function. If the
idempotent theorem (\forall x : \text{NonemptyT}. \text{join}(x, x) = x) doesn’t hold then the
proper type is indeed distinct from the type of containers with one member, as
the theorem in the table asserts. If the idempotent theorem does hold, however,
the proper type is the same as the non-empty type, as the subtype relation in
the table asserts.

We take this plethora of subtypes even further. Every constant introduced by
a scenario has its own type. In the Number scenario the zero has the zero type
and the constant one has the unity type. This allows us to set up lattices of types
piecemeal. When we first introduce the constant one its type will be a subtype
of the non-zero type. Later, if we introduce negative numbers its type will be a
subtype of the positive numbers. Later still we might need to distinguish even
and odd numbers and the zero type would then become a subtype of even and
the unity type a subtype of odd.

The motive for this is to represent as much theory as possible within the
type system, since that is cheap, and leave the predicate calculus to express only
that which can’t be expressed using types. However, the user has no control
over what is expressed in types. Whenever he gets a function or theorem from a
scenario any types that are needed are brought in automatically and all subtype
relations between those types are introduced into his work. This is done because
the subtype relations are part of syntax of the theorem, not separate independent
theorems.

\footnote{We distinguish between the normal join function, which accepts any T and the join-
proper which accepts only non empty T so that in theorems and refinements we can
express any container value as one of three mutually exclusive cases.}
4 The Standard Scenario

When I said, above, that the type system has no built-in types I was being, to borrow a term from Yes Minister, economical with the truth. The type system itself has no built-in types, but the CARE2 system does. There is a standard scenario which has some special properties. These properties include:

- the user cannot instantiate it, it is instantiated exactly once for every project
- all the operators of the scenario are available to all projects, but the user can’t change them in any way
- the user does not have direct access to any of the types it introduces, in particular, the user can’t create operators which take these types as arguments and can’t create functions which yield values of these types
- the semantics of the types are not provided by theorems, but are built-in to the operations of the system.

In some cases the type system is mis-used to enforce some syntactic constraint. There is, for example, a type which provides a context in which all the user can do is define a new variable. Others are more like genuine types. The predicate type gives a context in which the user can create formulae representing predicates, and the system provides operators such as ‘and’, ‘or’ and ‘not’ to combine them. By special dispensation the user can create operators which give this type as a result\(^3\) and so can create predicates in whatever theory is needed.

Another special type is called the universal type. All the types that the user introduces are automatically made a subtype of the universal type, so that it provides a context in which any user type is acceptable. This is used, for example, in specifications of fragments, where the user must be able to specify values of any accessible type. The universal type is also the type of the arguments of a built-in operator, equality. Thus, the user can assert the equality or otherwise of any values that the user can create.

Between the three of them, the variable type (as used in quantifiers), the predicate type and the universal type, together with the operators of those types, the standard scenario provides everything that is necessary to express anything in the first order predicate calculus with equality.

5 Why Not Use Z’s Type System?

That seems like a good idea. Z is rapidly becoming the premier specification notation in formal development and the user of CARE2 must necessarily be familiar with it. However, the goals of Z’s type system and ours are very different.

In Z, a type is described as a set known to contain a value\(^6\). Once we have said that types are sets, types take on a whole host of properties. In the context

\(^{3}\) So far as the user is concerned functions are operators which yield user’s types and predicates are operators which yield the predicate type and both can take only the user’s types as arguments and the user can’t create any other kind of operator.
of Z we need to do this. The intended reader of a Z specification is a human being and the purpose of the specification is to provide a model of an intended product for the human being to judge. In this case the reader is king. The writer must use notations and concepts that are familiar to the readers and the readers must expect that the notations and concepts used in one specification will be used in many others. Furthermore, the writer must try to say as much as possible to assist the readers in their judgment.

In CARE2, however, the reader is not a human being, it is a machine. Now the writer is king. In specifying something the writer is not providing a model, he is stating the required behaviour, neither more nor less. If anything further can be deduced from the specification, it can be deduced only from the specification. The writer must try to say as little as possible so as to constrain the subsequent implementation as little as possible.

In the subsequent implementation the writer is, of course, creating a model of the behaviour of the system, but the model is very different in nature and purpose from that created as the original specification. And establishing that difference, that is what CARE2 is for.

6 Where To From Here

There are two classes of users of CARE2. There is the journeyman software engineer, who is the one we have talked about above, and there is the scenario designer. The facilities available to the former are a subset of those available to the latter. For example, the engineer can only introduce new functions into the system by instantiating a function from a scenario, but the designer can create new functions in a scenario. However, when working in one scenario the designer often needs to draw material from another. In this case the designer is just as constrained as the engineer and can only introduce material by instantiation of the other scenario.

So far we have created five scenarios, the Number and Container scenarios and the Standard scenario referred to above, an Enumeration scenario which provides counter-like things which can’t be expressed easily as numbers and a scenario which provides tuples and discriminated unions similar to the disjoint union of Z. This last isn’t a lot of use to the engineer, since the theorems available allow him only to take things apart and put them back together; it is intended mainly for the scenario designer who can add other theorems and operators to make more useful types out of them.

(The Enumeration scenario has, at various times during the development been part of the Number scenario and at other time separate. They really should be together since there are many cases where the engineer will need one or the other but doesn’t need to decide which at once. I suspect that the whole library of scenarios will go through another complete re-write before I am satisfied and that these two scenarios will be unified once more.)

(An enumeration type is like a number type except that there is no left identity for addition, so you can’t do arithmetic within the type. It is useful for
things like dates, where it does not make sense to add 1st January, 1998 to 2nd March, 1998 to get 3rd April, 1996. On the other hand, you can add an interval of 60 days to 1st January to get 2nd March, so the user might have an addition operator with a signature like $\text{Date} + \text{Days}$ : $\text{Date}$. Enumerations can be bounded and the special case of an enumeration with two elements gives a type useful for recording boolean algebras. The maximum and minimum operators give ‘or’ and ‘and’ and an end-around increment operator gives ‘not’. However, such a type is quite distinct from the built-in predicate type.

The implementation has progressed to the stage where the designer can create new scenarios and populate them with material for the engineer to use, with the type system working as described above. Still missing is a mechanism to introduce induction theorems and functionals (functions which take functions as arguments and return functions as results). This shows a limitation of the present type system; without a function type constructor we can’t express such objects. CARE once had a type system that could express them, based on [1], but it had the same expressive power, and cost, as the predicate calculus and, since we already had everything we needed to use the predicate calculus, we weren’t winning anything and abandoned it. The present system handles the easy cases of type-like predicates and allows the full predicate calculus mechanisms to handle what remains. It turns out to be a worthwhile division of labour, but it does mean we have to find a special mechanism to express things like functionals.

We will have the same problems and expect to solve them when we implement the means for designers to create refinements for the engineers to use, which is the next thing we will be tackling in the development of CARE2.

References

5. Peter A. Lindsay and David Hemer. Using CARE to construct verified software.
A Proof Framework with IO Regular Expressions for Dataflow Networks

Yuji Hayashi,¹ Tadashi Yamaguchi² and Eiichi Miyamoto³

¹ Faculty of Business Administration and Information Science
   Hokkaido Information University, Ebetsu, 069-8585 Japan
   hayashi@do-johodai.ac.jp
² Division of Common Subjects (Mathematical Science)
   Muroran Institute of Technology, Muroran, 050-8585 Japan
   yamagut@oyna.cc.muroran-it.ac.jp
³ Graduate School of Engineering
   Hokkaido University, Sapporo, 060-8628 Japan
   miya@complex.eng.hokudai.ac.jp

Abstract. We present an expression form (IO regular expression) for streams and processes in a dataflow network, and a proof framework using the forms for a network property. Using IO regular expressions enables us to represent both streams and processes without using recursions, making it easier to substitute streams into processes. Our proof framework is based on the result that the solution stream vector of a dataflow network is the least fixed point of its equation. The proof is proceeded by assuming an input stream expression on each channel, producing an output stream expression on each channel and proving the equivalence of the input and output stream for the same channel. In this paper we discuss the semantics of IO regular expressions and show some examples to validate properties of streams in a dataflow network.

1 Introduction

A dataflow network [9,12] (later referred to as just network) is a model constructed with processes connected by channels. In this model, processes communicate through channels asynchronously with each other. Processes can memorize states, and can be viewed as black boxes that transform input streams into output streams. Dataflow networks are included in simplified models of the structured analysis [4]. Moreover several programming systems implementing dataflow networks have been developed [10,13]. A stream is an effective object to model a data relation within problems to be analyzed. There is a stream representation in functional programming [3], but the representation is conventional head/tail construction. It is hard to prove some properties for streams in such representations [16]. In this paper, a representation form for a dataflow network is introduced. This representation, IO regular expression, expresses both streams of channels and behaviors of processes. An IO regular expression is similar to a regular expression for a finite state machine and it includes terms that express static relations for the data on each channel. It is characteristic that an IO
regular expression represents iterations without any recursions. Formal methods
(Z [17], VDM [8]) express system states statically as relations or sets, but do
not express dynamic state transitions with the passage of time. An IO regular
expression is able to express dynamic state transitions.

Several proof methods for properties of a dataflow network have been re-
ported (Kahn [9], Nguyen [15]). There has been less research concerning asyn-
dataflow network are the fixed point stream vector of the equation that ex-
presses the relation within the network. Kahn presented a simple example for a
proof method using induction. However, in this method there is no formalization
for streams, so a general proof can not be shown. Using our expression (IO reg-
ular expression), the proof can be shown step by step, calculating each stream
for each channel. Nguyen [15] proved properties of networks using a representa-
tion based on temporal logic. Temporal logic [5] expresses logical relations but
not data streams. In addition, in temporal logic, static relations and dynamic
relations are mixed. An IO regular expression expresses streams and relations of
streams, with static data relations and dynamic data occurrence order being ex-
pressed separately. The authors’ proof method is progressed as follows. Initially,
processes are expressed with IO regular expressions. Secondly, input streams are
supposed with IO regular expressions for each channel, and output streams are
made from each process being substituted by the input streams. Finally, the
equivalence of an input stream and output stream for each channel is shown. If
a vector of calculated streams is the least fixed point, then some properties are
proven from the solution stream vector. Kahn showed that the least fixed point
is the solution stream vector of the network if the process behaviors of a network
are deterministic. In this paper, we consider the condition for an IO regular ex-
pression to be deterministic. With regard to the semantics of the proof basis, the
mathematical meaning of an IO regular expression is a relation of stream sets.
An IO regular expression for processes means a relation between input stream
sets and output stream sets.

In Section 2, a dataflow network is introduced, and an IO regular expression
is defined. In Section 3, the meaning of an IO regular expression is defined as
the relation of stream sets. In Section 4, meanings of substituting streams into
processes are considered. A proof method using the IO regular expression is in-
troduced in Section 5. In Section 6, two examples are shown: One demonstrates
a simple process communication, the other a distributed algorithm. Section 7
treats the determinacy conditions of IO regular expressions. Finally, in Section
8, some difficulties involving this method are discussed.

### 2 IO Regular Expressions

In this section, a dataflow network is introduced and an IO regular expression
is defined.
2.1 Processes and Channels in a Network

A dataflow network contains channels and processes (see Fig. 1). A channel is a directed arrow that is a path for data items. We call a unit data to be read (or written) on a channel an item. Each channel has an entry process and/or an exit process (these must not be the same process). An item enters the channel when written to the channel by the entry process, and comes out when read by the exit process. Thus a channel is an unbounded FIFO buffer memory. A channel that has no exit process is called a sink channel, and one that has no entry process is called a source channel. A finite (or infinite) sequence of data items is called a stream. Processes have one or more input or output channels, and may convert (during some computations) the input streams on input channels into the output streams on output channels. Processes can have a special channel that is called a local channel and is used to memorize data within the process. Such a channel has no exit process, similar to a sink channel, but is not able to be accessed from out of the process. A local channel is treated as an output channel for the process.

2.2 IO Regular Expression

As an expression of processes and streams, we define below an IO regular expression. An IO regular expression is called a process expression when it expresses the behavior of a process, and is called a stream expression when it expresses the stream on a channel.

Definition 1.

\[
\langle\text{operator}\rangle ::= + | - | \times | \div | \mod \\
\langle\text{arith element}\rangle ::= \langle\text{constant}\rangle | \langle\text{variable}\rangle | \langle\text{function}\rangle | (\langle\text{arith expression}\rangle) \\
\langle\text{arith expression}\rangle ::= \langle\text{arith element}\rangle | \langle\text{arith element}\rangle \langle\text{operator}\rangle \langle\text{arith element}\rangle
\]

\langle\text{constant}\rangle contains integers, strings or symbols representing invariant values. \langle\text{variable}\rangle contains names for channels (explained below). \langle\text{function}\rangle contains function calls like abs(z), sqrt(z) that are computable.

\[
\langle\text{compare\ operator}\rangle ::= \leq | \geq | \neq | > | < | = \\
\langle\text{conj expression}\rangle ::= \langle\text{compar expression}\rangle \langle\text{compare\ operator}\rangle \langle\text{compar expression}\rangle \\
\langle\text{logical element}\rangle ::= \langle\text{compar expression}\rangle | (\langle\text{IO relation}\rangle) \\
\langle\text{IO relation}\rangle ::= \langle\text{cong expression}\rangle \langle\text{logical\ element}\rangle \\
\langle\text{IO relation}\rangle ::= \langle\text{cong expression}\rangle \langle\text{cong expression}\rangle \langle\text{cong expression}\rangle \\
\langle\text{primary term}\rangle ::= [\langle\text{IO relation}\rangle]
\]
Each symbol not defined in detail is assumed to have the usual mathematical meaning. Priority orders of operators are also obeyed under the usual rules of logic. 

\[ \text{\textit{IO relation}} \]  is a logical formula that expresses relations between input items and output items at one action of the process. It might include a \( \text{\textit{variable}} \).

\( \text{\textit{variable}} \) expresses an item on input channels or output channels, possibly in a present or past state as follows.

- \( \alpha? \): Where \( \alpha \) is a name of an input channel. This is called a present input. This variable expresses an item that is read from channel \( \alpha \). The item is appended to a past stream on channel \( \alpha \).

- \( \beta \): Where \( \beta \) is a name of an output channel that might be a local channel. This is called a present output. This variable expresses an item that is written to channel \( \beta \). The item is appended to a past stream on channel \( \beta \).

- \( \overline{\alpha}, \overline{\beta} \): The data refers to the latest item of the past stream on the channel. This is called the latest input (or output). We can refer past data without particular commands for memorizations.

- \( \gamma \): Where \( \gamma \) is any name. This is an auxiliary variable that is used to memorize data within a stream. It is also called a present output. In a process expression, data can be memorized by local channels, so this symbol is used for streams only.

A term is constructed by primary terms with operations (iteration, sequence and selection) as follows.

**Definition 2.**

\[
\begin{align*}
\langle \text{factor} \rangle & ::= \langle \text{primary term} \rangle | \langle \text{factor} \rangle \langle \text{factor} \rangle \\
& | \langle \text{primary term} \rangle^* | \langle \text{primary term} \rangle^\omega \\
& | \langle \text{primary term} \rangle^\omega | \langle \text{primary term} \rangle^\omega \\
\langle \text{seq factor} \rangle & ::= \langle \text{factor} \rangle \langle \text{seq factor} \rangle \\
\langle \text{term} \rangle & ::= \langle \text{seq factor} \rangle | \langle \text{seq factor} \rangle \mid \langle \text{seq factor} \rangle \\
\langle \text{IO regular expression} \rangle & ::= \langle \text{term} \rangle
\end{align*}
\]

The meanings of iteration (\( \ast \) and \( \omega \)), sequence and selection (\( \mid \)) (listed in priority order) are similar to the regular expression of the finite state machine (and the \( \omega \)-machine [14]). The precise meaning of these operations is explained in the next section.

**Example 1.** The following is an example of a process that has input channel \( \text{in} \) and output channel \( \text{out} \). This process terminates when the input value equals 0 after the iteration of writing an item that has just been read from channel \( \text{in} \).

\[
[\text{in?} > 0 \land \text{out} = \text{in?}]^* [\text{in?} = 0].
\]
Example 2. The following is an example of a process that produces a finite sequence \((1, 2, \ldots, 100)\) on channel \(\text{out}\).

\[
\text{[out} = 1\text{]} \quad \text{out} = \overline{\text{out}} + 1 \land \text{out} \leq 100^* .
\]

Definition 3. An IO regular expression is called independent when all variables that express the latest input or output have the corresponding variables (present inputs or present outputs) within it.

For example, the term \([\overline{\text{out}} > 0]\) \([\text{out} = \overline{\text{out}} + 1]\) is not independent because it has no corresponding variables to \(\overline{\text{out}}\) within the term.

3 Semantics of IO Regular Expressions

An IO regular expression, although similar to a temporal logic formula, means relations of stream sets on channels.

3.1 N-ary Relations Represented by IO Regular Expressions

In this paper, let \(\Delta\) (and \(\Delta\) with a subscript) be any (finite or infinite) sets of items.

Definition 4. Let \(\vdash\) be a special symbol that is not an element in \(\Delta\) and expresses the boundary between past and present item sequences. Then define

\[
\Delta^l = \Delta^* \vdash \Delta^\infty ,
\]

where \(\Delta^\infty = \Delta^* \cup \Delta^\omega\) and \(\Delta^\omega = \lim_{k \to \infty} \Delta^k .\)

Definition 5.

\[
\Delta^l = \Delta^l_1 \times \Delta^l_2 \times \ldots \times \Delta^l_n
\]

Where \(\Delta_i\) is a set of items on channel \(i\). \(n\) is the total number of channels in a network. An element of \(\Delta^l\) is called a stream vector.

Let \(X\) be any stream vector, and \(X_{(i)\{C\}}\) be an element of channel \(i\) for \(X\) \(D\). For a channel set \(Ch = \{c_1, c_2, \ldots, c_k\}\) \(C\) define

\[
X_{(C\{h\)} = (v_{c_1}, v_{c_2}, \ldots, v_{c_k}) \text{ if } X = (v_1, v_2, \ldots, v_{c_2}, \ldots, v_{c_k}, \ldots, v_n). \]

We introduce symbols of an empty item sequence \(e\) and an empty stream set \(\Delta^H\) and empty stream vectors set \(\Theta\) as follows.

Definition 6.

\[
(\text{empty item sequence}) \quad e \text{ means the empty sequence of item (set representing empty streams)} \quad \Delta^H = \Delta^* \vdash
\]

\[
(\text{set representing empty stream vectors}) \quad \Theta = \Delta^H_1 \times \Delta^H_2 \times \ldots \times \Delta^H_n
\]
Definition 7. Suppose $x, y \in \Delta'$. Define ’$x$ is a prefix of $y$’ as follows:

$$x \preceq y \text{ if } \exists w \in \Delta^\infty : y = xw$$

For $U, V \in \Delta'$, define $U \preceq V$

$$\text{if } U(i) \preceq V(i) \ (1 \leq i \leq n).$$

For $A, B \in \Delta'$, define $A \preceq B$

$$\text{if } \forall V \in B : \exists U \in A : U \preceq V.$$ 

In an IO relation of a primary term, the present input item ($\alpha$), and the present output item ($\beta$) are the right side items to $\vdash$. The latest input and output are the left side items to $\vdash$. The IO relation $h$ is expressed precisely as $h(e_1, d_1; \ldots; e_n, d_n)$, where $e_i$ is the latest input (or output) and $d_i$ is the present input (or output) for channel $i$ in a network with $n$ channels. Both of $e_i$ and $d_i$ may be empty. For example, suppose that a network has three channels ($\text{ina}, \text{inb}, \text{outa}$). If an IO relation $h$ is

$$\text{ina} > 0 \land \text{ina} = \text{outa} + \text{ina}$$

then the IO relation is presented as follows.

$$h(\text{ina}?, \text{ina}?, \text{ina}?, \text{ina}?, \text{outa})$$

An IO relation whose variables are replaced by items means a logical expression as follows.

$$h(3, 4; 4; 4, \text{outa}) \equiv 4 \land 4 = \text{outa} + 3$$

For a primary term $t = [h(\gamma_1, \delta_1; \ldots; \gamma_n, \delta_n)]$, $n$-ary relation of meanings for $t$ is represented as $\tau(t) \subseteq \Delta'$ and is defined as follows, where $n$ is the number of channels in the network.

Definition 8. $\tau([h(\gamma_1, \delta_1; \ldots; \gamma_n, \delta_n)])$ is a set of stream vector $X$ whose elements $X_{(i)}$ satisfy the following.

$$X_{(i)} = u_i e_i \vdash d_i \ (i = 1, 2, \ldots, n),$$

where $e_i = e \ i f \ \gamma_i = \text{empty},$

$\quad d_i = e \ i f \ \delta_i = \text{empty}$

and $h(e_1, d_1; \ldots; e_n, d_n) = \text{true}.$

$\tau(t)$ means a set of stream vectors that occur on every channel until execution of $t$ is completed. The right side of $\vdash$ is an item that occurs in this primary term and the left side is an item sequence that possibly occurred before starting to execute $t$.

The following is a definition for a sequence of two streams.
Definition 9. For $c \in \Delta^* \vdash \Delta^*$, $d \in \Delta^* \vdash \Delta^\infty$ a sequence $c \cdot d$ is defined as follows.

\[
c \cdot d = \begin{cases} \vdash xy & \text{if } c = \vdash x \land d = \vdash y \\ \text{undefined} & \text{(otherwise)} \end{cases}
\]

Let $L,R$ and $S$ be any IO regular expressions in the following Definitions 10 to 13. We define meanings of a sequence of two IO regular expressions as follows.

Definition 10. Define

\[
\hat{\tau}(LR) = \{(l_1,l_2,\ldots,l_n) | l_i = x(i) \cdot y(i) \in \Delta^i, 1 \leq i \leq n \land x \in \tau(L) \land y \in \tau(R)\}.
\]

Then $\tau$ is defined using $\hat{\tau}$:

In case $R$ is non-input driven,

$\tau(LR) = \hat{\tau}(LR)$.

In case $R$ is not non-input driven,

$\tau(LR) = \tau(L) \cup \hat{\tau}(LR)$ (if $L$ is a primary term),

$\tau(LR) = \tau(XR) \cup \tau(YR)$ (if $L = X \cup Y$),

$\tau(LR) = \tau(X(YR))$ (if $L = XY$), and

$\tau(LR) = \tau(X^*) \cup \hat{\tau}(X^*R)$ (if $L = X^*$ or $L = X^\omega$).

Here a term $R$ is called non-input driven if

\[
\forall U \in C(R) : \forall \text{input channel } i \in In : U(i) \in \Delta^* \vdash .
\]

“In” is a set of input channels, $C(R)$ is the condition set to execute $R$ (see Definition 17) to be explained in more depth later. That is, the non-input driven expression shows that the evaluation is started without any present input item.

For an IO regular expression including a present input variable, it is possible that any given input item for the variable may never arrive at the channel. Thus the meaning of the expression should include prefix stream vectors up to the present input. But a non-input driven term is evaluated without waiting for any new input, so does not include any prefix in the meanings set.

The meaning of the selection is defined as follows.

Definition 11.

$\tau(R \cup S) = \tau(R) \cup \tau(S)$.

To define meanings of iterations, the following definition of the least upper bound is necessary.

Definition 12. For any infinite (or finite) sequence

\[
U_1 \preceq U_2 \preceq \ldots \preceq U_i \preceq \ldots
\]

$Z$ is called the least upper bound of $U_1, U_2, \ldots, U_i, \ldots$.
if (\forall i: U_i \leq Z) \text{ and } (\forall i: U_i \leq W \implies Z \leq W).

The least upper bound Z is expressed as

\[ Z = \lim_{k \to \infty} U_k. \]

For the iterations * and \( \omega \), the meanings are defined as follows.

**Definition 13.**

\[ \tau(L^*) = \begin{cases} \bigcup_{0 \leq k < \infty} \tau(L^k) & (\text{for stream expressions}) \\ \lim_{k \to \infty} \tau(L^k) & (\text{for process expressions}) \end{cases} \]

\[ \tau(L^{\omega}) = \lim_{k \to \infty} \tau(L^k) \quad (\text{for both stream and process expressions}) \]

where \( L^k = L^{(k-1)}L \) (for \( k = 1, 2, \ldots \)) and \( \tau(L^0) = \Theta \) (for \( k = 0 \)).

As per the definition above, \( L^* \) and \( L^{\omega} \) basically mean distinct stream sets. In stream expressions, * means a set of finite streams, \( \omega \) means a set of possibly infinite streams. In process expressions, * and \( \omega \) mean the same set possibly including an infinite stream. For example, consider an IO regular expression \( L = [x = 1]^* \). If \( L \) is a stream expression, then \( \tau(L) \) includes all streams with \( x \) satisfying

\[ \Delta^* \vdash \Delta^* \vdash 1, \Delta^* \vdash 11, \Delta^* \vdash 111, \ldots \]

If \( L \) is a process expression then \( \tau(L) \) means that the process outputs an infinite stream of 1 on \( x \),

\[ \Delta^* \vdash 111111\ldots \]

However, process expression \( R = [x? > 0]^* \) includes any prefix part of it, because it is not non-input driven. Then \( \tau(L) \subset \tau(R) \) is satisfied for the above stream expression \( L \), and the stream \( L \) could be substituted into the process \( R \), as will be explained in the next section.

**Definition 14.** For any IO regular expressions \( t, s \):

\[ t \cong s \text{ if and only if } \tau(t) = \tau(s). \]

### 4 Meanings of Stream Substitution

In this section we investigate the substitution of streams into process expressions. A set of streams on a channel is represented by a stream expression. The set of all streams able to be read through the channel is represented by a process expression. Only streams that are elements in both sets at the same time are read by the process. The following definitions on substitution includes these properties.
**Definition 15.** Let \( Ch \) be a set of channels of a network, \( s \) be a stream expression and \( p \) be a process expression that has an input (output) channel set \( In \) (Out). Let \( Els = Ch - (In \cup Out) \).

Define \( p(s/(In)) \) as a stream expression whose meaning set is a collection of \( Y \in \mathcal{A}' \) that satisfies the following.

\[
Y_{(In)} = X_{(In)}, Y_{(Els)} = X_{(Els)} \text{ and } Y_{(Out)} = P_{(Out)},
\]

where \( X \in \tau(s), P \in \tau(p) \) and \( X_{(In)} = P_{(In)} \).

Then substituting a stream expression into a process expression means calculating \( \tau(p(s/(In))) \).

This procedure is done as follows. Suppose a process has one input channel \( \text{in} \). Let \( t \) be a process expression of the process and \( s \) be a stream expression substituted into \( t \). Moreover, suppose that \( \tau(s)_{(in)} \subseteq \tau(t)_{(in)} \) and every primary term including \( \text{in} \) in \( s \) corresponds (one to one) to a primary term including \( \text{in} \) in \( t \). If the primary terms \([p(\text{in})]\) in \( s \) corresponds to \([g(\text{in})]\) in \( t \), then the primary term \([g(\text{in})]\) is changed into

\[
[p(\text{in}) \land g(\text{in})]
\]

This primary term may be changed by the usual logic rules, if necessary.

If the correspondence between \( s \) and \( t \) is not found directly, then it may be necessary to reform the stream expression and/or the process expression. When \( \tau(s)_{(in)} \subseteq \tau(t)_{(in)} \) is not satisfied, a stream expression corresponding to subset \( \tau(s)_{(in)} \cap \tau(t)_{(in)} \) should be substituted into the process.

**Example 3.** Substitute stream expression \( s = [\text{in} = 1]^* [\text{in} = 0] \) into process expression \( p = [\text{in} > 0 \land \text{out} = \text{in}^?]^* [\text{in} \leq 0] \). \( \tau(s)_{(in)} \subseteq \tau(p)_{(in)} \) is satisfied, so we get

\[
p(s/(\text{in}^?)) \cong [\text{in} = 1 \land \text{in} > 0 \land \text{out} = \text{in}]^* [\text{in} = 0 \land \text{in} \leq 0].
\]

For another case, substitute the stream \( r = [\text{in} = 3] [\text{in} = 4] \) into the same process expression \( p \). Then \( \tau(r)_{(in)} \subseteq \tau(p)_{(in)} \) is satisfied too. In this case we don’t find direct correspondence between \( r \) and \( p \). The meaning set of process expression \( p \) includes its prefix parts. \( \tau(p) \) includes a set comprising a sequence of two terms of \([\text{in} > 0 \land \text{out} = \text{in}^?]\), so we get

\[
p(r/(\text{in}^?)) \cong [\text{in} = 3 \land \text{in} > 0 \land \text{out} = \text{in}] [\text{in} = 4 \land \text{in} > 0 \land \text{out} = \text{in}].
\]

5 Proof for a Network

A network starts execution when at least one of the following states occurs.

1. Some process reads an item from some source channel.
2. Some process writes an item to an output channel without reading any item from input channels.

We assume that process expressions for each process are already verified.

Intentions of networks are generally represented in streams on channels. We want to validate that streams of the network satisfy some given properties. The properties are presented with an IO regular expression, and we call this expression a target expression.

We restrict, in this paper, the target expression to an expression representing stream properties on a particular channel, as in the following: Suppose that a network has source channels \((in_1, in_2, \ldots, in_k)\), and a sink channel is \(out_1\). Then the target expression \(f_1\) on \(out_1\) is generally the following relation.

\[
\tau(f_1) \subseteq \Delta_{in_1} \times \Delta_{in_2} \times \cdots \times \Delta_{in_k} \times \Delta_{out_1}
\]

We want to show that the meaning set of channel \(out_1\) is included in \(\tau(f_1)\).

Our method proves the correctness of the target expression after calculating the stream expression on the channel of the target expression.

When the execution of the network proceeds infinitely, ultimate streams on channels are either of the following.

1. Output streams on all channels are read by the exit processes (assumed an infinite long stream is also read).
2. When the execution of the network proceeds infinitely, there is a part of the stream left that is never read by the exit channel.

Our subject of proof is that the target expression is satisfied after the execution of the network is finished (or locked into an endless execution).

The procedure of our proof is summarized as follows.

1. Express streams on the source channels as stream expressions if there are some source channels.
2. Represent the target expression on the particular channel.
3. For each process, represent stream expressions to be read on input channels of the process. We call this the input stream expression for the channel.
4. For each process, substitute the above input stream expression into a process expression.
5. For each process, calculate stream expressions to be written on the output channel of the process from the above substituted expression. We call this the output stream expression for the channel.
6. For each channel (without source channels or sink channels), prove the following if necessary:

\[
\text{the input stream expression} \cong \text{the output stream expression}
\]

If this is satisfied on each channel, then the stream expression vector is the fixed point of an equation system for the network.

7. Prove the stream vector on each channel derived until step 6 is the least fixed point vector.
8. Lead the target expression from the derived stream vector until step 7.
6 Examples of the Proof

We illustrate our method by using the example by Kahn [9].

Example 4. Consider a network containing process $G,F$ and channel $x,y,z$. The process $G$ writes 0 on channel $y$ and 1 on channel $z$, and repeats an action that reads an item on $x$ and writes it on $x$, and reads an item on $x$ and writes it on $x$. The process $F$ repeats an action that reads an item on $y$ and writes it on $x$, and reads an item on $z$ and writes it on $y$. We want to prove that the stream on channel $x$ is an infinite sequence of 0 and 1 appearing alternatively.

We write down the proof procedure adding the proof step numbers that can be seen above, in steps 1 to 8.

```
  y

     F

     x

     z

     G
```

**Fig. 1.** A network with feedback.

step(2),(3): Stream on $x$(target expression $\text{fin}(x)$) is

$$\text{fin}(x) = [[x = 0] [x = 1]]^\omega$$

($\omega$ means an infinite stream because of being a stream expression.)

The channels of process $G$ is

```
input channel (x)
output channel (y z)
```

The process expression of $G$ is

$$g(x; y, z) = [y = 0 \land z = 1] [[y = x?] [z = x?]]^\omega .$$

The process $G$ may iterate forever to read $x$ and write $y,z$ ($\omega$ includes a case of a finite stream because of being a process expression.)

step(4): The expression of stream $x$ substituted into the process expression $g$ is

$$g(\text{fin}(x)/(x?)) \equiv [y = 0 \land z = 1] [[y = 0] [z = 1]]^\omega .$$

step(5): The stream expression of output on $y$ is

$$g(\text{fin}(x)/(x?)) | y \equiv [y = 0] [y = 0]^\omega .$$
step(5): The stream expression of output on \( z \) is

\[
g(\text{fin}(x)/(x?))_{|z|} \cong [z = 1] [z = 1]^w.
\]

The channels of process \( F \) is

<table>
<thead>
<tr>
<th>input channel</th>
<th>[y x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>output channel</td>
<td>(x)</td>
</tr>
</tbody>
</table>

The process expression of \( F \) is

\[
f(y, z; x) = [[x = y?] [x = z?]]^w.
\]

step(4): The expression substituted by streams \((y, z)\) into the process expression \( f \) is

\[
f(g_{(y?)}(y?), g_{(z?)}(z?)) \cong [x = 0] [x = 1] [[x = 0] [x = 1]^w].
\]

step(5): The stream expression of output on \( x \) is

\[
f(g_{(y?)}(y?), g_{(z?)}(z?))_{|x|} \cong [x = 0] [x = 1] [[x = 0] [x = 1]^w].
\]

step(6): Equivalence of the input stream on \( x \), \( \text{fin}(x) \), to the output stream on \( x \), \( f(g_{(y?)}(y?), g_{(z?)}(z?))_{|x|} \). These are both infinite \( 0,1 \) sequences (meaning sets are the same).

Then this implies,

\[
f(g_{(y?)}(y?), g_{(z?)}(z?))_{|x|} \cong \text{fin}(x).
\]

And \((g_{(y?)}(y?), g_{(z?)}(z?))_{|x|}, g(\text{fin}(x)/(x?))_{|y|}, g(\text{fin}(x)/(x?))_{|z|})\) can be proved to be the least fixed point stream vector as follows.

step(7): Suppose that a proper prefix of \( f_{|x|} = (01)^w \) is a fixed point, then there exists a non negative integer \( k \) and \( f_{|x|} = (01)^k D \) When \( (01)^k \) is given as an input stream to the process \( G \), the output streams on \( y, z \) must have total length \( k + 2 \); thus the output stream from \( F \) has length \( k + 2 \). This is a contradiction, so \((01)^k\) is not a fixed point. For other channels, the proof can be completed in the same way.

Example 5. Distributed algorithm (finding the maximum identification number [1]).

Consider a network that has \( n \) processes connected ringwise with channels. Let the processes be \( P_0, P_1, \ldots, P_{n-1} \), and each process \( P_i \) has an input channel \( x_{\text{read}(i-1)} \) and an output channel \( x_i \), (where \( \text{mod}(k) = k \mod n \)). Each process has a unique identification number \( \text{val}_i \). All processes have the same algorithm.

The simplest method for all processes to get the maximum identification number by exchanging data with each other through channels is as follows. First, each process \( P_i \) writes its own ID number on channel \( i \) within a finite time period, then repeats writing read data until reading its own ID number. During the repetition, each process memorizes the intermediate maximum number in a local channel \( mx \).

The channels of process \( P_i \) are the following.

---

The rest of the text is not provided.
Fig. 2. A network for finding the maximum ID number.

<table>
<thead>
<tr>
<th>input channel</th>
<th>$(x_{md(i-1)})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>output channel</td>
<td>$(x_i)$</td>
</tr>
<tr>
<td>local channel</td>
<td>$(mx)$</td>
</tr>
</tbody>
</table>

In the following, $c'_i$ is an auxiliary variable (they have the same name in all channels, but they are distinct variables) in a stream expression.

Process expression $P_1 F$

\[
\left[ x_i = val_i \land mx = val_i \right] \left[ x_{md(i-1) \neq val_i} \land mx = \max(mx, x_{md(i-1)}) \land x_i = x_{md(i-1)} \right] \left[ x_{md(i-1)} = val_i \right]^* \]

We want to show that after the network’s execution has finished, each local channel $mx$ saves the maximum value. It is necessary to show that the variable $mx$ becomes the following stream in each process $P_i$.

Target expression:

\[
\left[ c'_i = i \land mx = val_i \right] \left[ c'_i \neq i \land mx = \max(val_k \mid k \in wh(c'_i, i)) \right] \left[ c'_i = \text{md}(c'_i - 1) \right]^* ,
\]

where $\text{wh}(a, b)$ is the following set.

$\text{wh}(a, b) = \{a = \text{md}(a), \text{md}(a + 1), \ldots, \text{md}(a + k) = b\}$ \hspace{1cm} (0 \leq k < n)

Assume the stream on input channel $x_{md(i-1)}$ of process $P_i$ as

\[
\left[ c'_i = i \left[ x_{md(i-1)} \neq val_i \land x_{md(i-1)} = val_i \land c'_i = \text{md}(c'_i - 1) \right] \left[ x_{md(i-1)} = val_i \right]^* \right]
\]
and substitute this stream into the process expression of $P_i$. Then we get

$$\begin{align*}
&\left[ c^*_j = i \\
&\land x_i = \text{val}_i \\
&\land \text{mx} = \text{val}_i \\
&\land x_{\text{md}(i-1)} \neq \text{val}_i \\
&\land x_{\text{md}(i-1)} = \text{val}_i \\
&\land c^*_j = \text{md}(c^*_j - 1) \\
&\land \text{mx} = \max(\text{mx}, \text{val}_i) \\
&\land x_i = x_{\text{md}(i-1)}
\right]^* \ \\
&\left[ x_{\text{md}(i-1)} = \text{val}_i \right]
\end{align*}$$

After eliminating $x_{\text{md}(i-1)}$ from this expression, we get a stream on channel $x_i$ as follows.

$$\begin{align*}
&\left[ c^*_j = \text{md}(i + 1) \right] \\
&\land x_i = \text{val}_{\text{md}(i+1)} \\
&\land c^*_j = \text{md}(c^*_j - 1)
\right]^* \ \\
&\left[ x_i = \text{val}_{\text{md}(i+1)} \right]
\end{align*}$$

This is the input stream of process $P_{\text{md}(i+1)}$, and is the same form as the input stream of process $P_i$ (except subscripts). Thus we can substitute it into the process $P_{\text{md}(i+1)}$ similarly, proving it is an element of a fixed point stream vector.

Proof of being the least fixed point:

Suppose that a proper prefix of the stream derived above is also the least fixed point stream, then after completing one more cycle of the ring, the supposed stream becomes a stream appended by one ID. This is a contradiction as the stream is no longer a minimum.

A stream on the local channel $\text{mx}$ has the following expression.

$$\begin{align*}
&\left[ c^*_j = i \\
&\land \text{mx} = \text{val}_i \\
&\land c^*_j = i \\
&\land \text{mx} = \max(\text{mx}, \text{val}_i) \\
&\land c^*_j = \text{md}(c^*_j - 1)
\right]^* \\
&\left[ x_i = \text{val}_{\text{md}(i+1)} \right]
\end{align*}$$

It can be proved by induction that this expression satisfies the target expression, so the proof is completed.

7 Condition Sets and Determinacy

In this section we investigate the determinacy of IO regular expressions.

**Definition 16.** Let $x = u \vdash v (u \in \Delta^*, v \in \Delta^\infty)$.

Then we define $\bar{x}$ and $\bar{\bar{x}}$ as follows.

$$\bar{x} = u \vdash a_0 \text{ and } \bar{\bar{x}} = u \vdash$$

where $a_0 = \begin{cases} 
\epsilon & (i f v = \epsilon) \\
\delta & (i f v = aw, a \in \Delta).
\end{cases}$

A condition set $C(t)$ for $t$ is defined below. This set indicates the set of items' vectors to start executing the term.
**Definition 17.**

\[ C(t) = \{ (\chi_1, \chi_2, \ldots, \chi_n) \mid X \in \tau(t) \text{ and } \chi_i = \sum_j \chi_{i(j)} \ (i \text{ if } i \text{ is an input channel}), \]
\[ \chi_i = \sum_j \chi_{i(j)} \ (i \text{ if } i \text{ is not an input channel}) \}. \]

For example, let in and out be an input channel and an output channel respectively, such that

\[ t = [\text{in} > 0 \land \text{out} = \text{in}] * [\text{in} = 0 \land \text{out} = \text{in}] . \]

Then we get

\[ C(t) = \{ (u \vdash h, v \vdash h) \mid h \geq 0 \text{ and } u \in \Delta_{in}^* \text{ and } v \in \Delta_{out}^* \}, \]

where the first of the 2-tuple is for channel in, the second is for channel out. This set means that if the present input for channel in is a negative number then the term t does not execute it.

Generally, in an IO regular expression, the plural distinct outputs are possibly produced from the same state of present and past streams. We will limit our use of process expressions to representing deterministic behaviors. The condition of IO regular expressions to be deterministic are considered below.

**Definition 18.** A pair of streams u, v are called separable if u, v are not a prefix of each other, and we write

\[ u \otimes v . \]

If \( \forall U \in C(t_1) : \forall V \in C(t_2) : \exists i (1 \leq i \leq n) : U_{(i)} \otimes V_{(i)}, \) then the terms \( t_1, t_2 \) are called separable and we write

\[ t_1 \otimes t_2 . \]

IO regular expressions of a process are assumed to satisfy the following conditions in each form (primary term, selection, iteration and sequence) to behave deterministically.

1. **primary term:** \( \forall X, Y \in \tau(t) : X \neq Y \Rightarrow (\exists \text{ input channel } k : \sum_k \otimes \sum_k) \text{ or } (\exists \text{ output channel } k : \sum_k \otimes \sum_k) . \)

   If two stream vectors for a process expression are distinct, then there must be distinct items in either the latest output, present input or latest input.

2. **selection:** If \( t_1 \sqcup t_2 \) then \( t_1 \otimes t_2 . \)

3. **iteration (* and \( \omega \):** For a term \( t^* \text{ (or } t^{\omega} \), t satisfies the following.

   if \( t = s_1 \sqcup s_2 \sqcup \ldots \sqcup s_m \) and \( s_k \) satisfies \( s_k = t_1 t_2^* \), then \( t \otimes t_2 . \)
4. sequence: If a term is \( (s_1 \sqcup s_2 \sqcup \ldots \sqcup s_m) t \) and \( \exists k : s_k = t_1 t_2^* \), then \( t_2 \otimes t \).

We call (1)-(4) determinacy conditionsD It is known that a dataflow network whose behavior is not decided uniquely has the undesired property [11, 12] of not being monotone. If an IO regular expression satisfies the determinacy conditions, then the output is decided uniquely for the same present input, latest input and latest output.

Suppose \( In(Out) \) is an input(output) channel set of process \( P \), as follows: \( In = \{ i_{n_1}, i_{n_2}, \ldots, i_{n_k} \} \), \( Out = \{ o_{m_1}, o_{m_2}, \ldots, o_{m_m} \} \), and let

\[
\Delta'_{(In)} = \Delta'_{in_1} \times \Delta'_{in_2} \times \ldots \times \Delta'_{in_k},
\]

\[
\Delta'_{(Out)} = \Delta'_{out_1} \times \Delta'_{out_2} \times \ldots \times \Delta'_{out_m},
\]

\[
\Psi = \{ G \mid G \subset \Delta'_{(In)} \}, \text{and}
\]

\[
\Phi = \{ H \mid H \subset \Delta'_{(Out)} \}.
\]

The process expression \( p \) of the process \( P \) represents a function from the set \( \Psi \) of stream vector sets mapping into the set \( \Phi \) of stream vector sets.

\[
p : \Psi \rightarrow \Phi.
\]

Meaning of substituting by a tuple \( s \) of stream expressions into the process expression \( p \) is \( \tau(p(s/\{In\}) ) \subset \Delta' \). This is a relation of stream sets on all of the channels in the network, though the above function \( p \) is given as a relation whose channels are restricted to \( In \cup Out \) instead of all channels in the whole network.

\( \Psi, \Phi \) are cpo (complete partially ordered set) [6] under \( \preceq \), so we have the next proposition (proof is omitted).

**Proposition 1.** If a process expression satisfies determinacy conditions (1) to (4), then it is continuous as a function of type \( \Psi \rightarrow \Phi \).

It is known that a continuous function over cpo has a least fixed point [6,9]. This result is the base of our proof framework. Actually the main parts of the proof are the procedure to find a fixed point vector of streams.

8 Evaluation for the Proof Framework

There are several difficulties in our proof framework. This is only a framework, so neither the transformation of expressions nor the proof of the equivalence between expressions are formalized yet. Each step should be carried out with care considering the semantics of the IO regular expressions.

Difficulties may occur in four phases: (a) in the assumption of the input stream expression on some channels, (b) in the substitution of streams into a process expression, (c) in the proof of equivalence between an input stream expression and an output stream expression on the same channel, and (d) in the leading step of the target expression from the calculated stream expressions.
In phase (a), we have to find some input stream expressions to be assumed that represent sets of streams produced during the execution of the network. This phase is similar to the finding of loop invariants in Hoare logic [7]. There is no general method for it, but the expression might be a good candidate if it is represented such that the occurrences of its output variable correspond (one to one) to the occurrences of the same variable in the process expression. In phase (b), we should confirm that there is one to one correspondence between the occurrence items (for a variable $\beta$) on the input stream expression $s$ and the occurrence items (for a variable $\beta$) on the process expression $t$. If the one to one correspondence is not found easily, the nature of the set $\tau(s)_{(\beta)} \cap \tau(t)_{(\beta)}$ should be investigated. Then the stream expression $s$ and/or the process expression $t$ must be arranged to make the existence of the one to one correspondence in $\tau(s)_{(\beta)} \cap \tau(t)_{(\beta)}$ clear. Sometimes the input stream expression and/or the process expression must be shortened. In phase (c), an IO regular expression has semantics of relations of stream sets rather than logical values, so there is no deductive system to be able to prove equivalence between two IO regular expressions yet. We may transform stream expressions into relation expressions of semantics sets, then we may be able to prove the equivalence between two relations with the properties of the set theory. In the case of plural input channels for the process, we sometimes have to represent streams on these channels as a combined stream expression to be substituted into the process expression. In phase (d), if the target expression is expressed in an IO regular expression, then this phase is similar to the above phase (c). Otherwise the target expression is expressed as another logical formula, and we may be able to treat the expression as an output condition in Hoare (or Dahl)'s proofs [7, 2].

9 Concluding Remarks

In this paper, a representation (IO regular expression) for streams and processes in a dataflow network has been reported. One advantage of the representation is that both a stream and a process can be expressed in the same form, so problems modeled by dataflow networks are represented by IO regular expressions. But the present version of our model covers only networks without recursion. We have shown the proof framework with the expressions for some properties of streams on channels. If this expression is used as a specification form, then the proof means validation of the specification. An IO regular expression is expected to be used also as an implementation language. Furthermore extensions of the expression to introduce data types, recursions and synchronous communications are expected. We plan to publish these works at a later date.
References

Can Component Object Model (COM) be Formalized? - The Formal Semantics of the COMEL Language

Rosziati Ibrahim and Clemens Szyperski
School of Computing Science
Queensland University of Technology
Brisbane, Australia
ibrahim@fit.qut.edu.au c.szyperski@qut.edu.au

Abstract. This paper presents an approach to formalize COM. Despite its importance, COM still does not have a formal specification. In order to understand the COM’s informal rules better, the COMEL language is being introduced. We formalized some of the important COM’s rules and present COMEL’s abstract syntax, formal semantics and subject reduction theorem.

1 Introduction

Microsoft’s OLE (Object Linking and Embedding) provides an application integration framework for Microsoft Windows. OLE rests on the Component Object Model (COM), which specifies a programming language independent binary standard for object invocations, plus a number of interfaces for foundational services.

COM is all about interoperability of independently deployed components and is used to develop component-based software. COM is language independent. The component software can be developed by independent vendors. Extensions to component software can also be developed and integrated by the client. Currently, COM offers a set of informal rules in order to form a standard binary level of building interoperable software. This set of informal rules is discussed in [2, 4, 8, 9, 12]. However, COM does not have a formal specification. COM’s rules are complex and subtle, making it worthwhile to formalize COM.

A model language is introduced in order to formalize COM. Since COM itself is language independent, the language introduced here just takes an exemplary role. This example language for COM is called COMEL (Component Object Model Extended Language, pronounced cho-mell). This paper will discuss the motivational aspects of why COM needs to be formalized, the construction of the formal semantics of COMEL in relation to COM rules, the abstract syntax and the formal semantics of the COMEL language and finally the subject reduction theorem and soundness theorem for COMEL.
2 Motivation - Why COM?

A component in a COM environment corresponds to a module containing multiple classes. A COM object is an instance of such a class. COM objects have state and an identity. All interaction with a COM object is through COM interfaces. A COM class can support any number of interfaces without merging them into a single name space (as Java for example does). A COM interface has any number of methods, but no fields. The usual notion of subtyping is replaced by that of supersets of interfaces and by the separation of implementations (classes) from interfaces. Implementation inheritance across components is not supported in COM (some of this is added in COM++; COM++ is the new generation of COM). COM supports only single interface inheritance.

As said before, COM is all about interoperability of independently deployed components. In order to achieve interoperability, a system needs to be independently extensible. Independently extensible systems allow components to be plugged into the running system when needed [14]. COM's lack of support for implementation inheritance is an advantage for developing independently extensible systems because it avoids overly tight coupling between code from different vendors.

Versioning of interfaces is an important problem in independently extensible systems. COM interfaces and their specification are immutable - once made public, they can never be changed. The QueryInterface function requires each interface to have a globally unique identifier (GUID). Therefore, if an implementor wants to add new functionality to the interface, he or she is required to define a new interface and a new GUID needs to be assigned to this new interface. A component can then simultaneously support the old and the new interface.

The three special properties of COM discussed above (any number of interfaces per class, no implementation inheritance, and immutable interfaces) set COM off from other object models and are the major motivation for formalizing COM.

COM is one of the few examples of available technology to build independently extensible systems. The methodology and theoretical aspects of independently extensible systems have not yet been sufficiently addressed [14]. Therefore, it is worthwhile to build a formal model for independently extensible systems. Starting with a formal model of COM, the hope is that this can then be generalized to a formal model for independently extensible systems.

One of the very important COM rules is that all COM interfaces must derive directly or indirectly from the IUnknown interface. This means that any interface implemented on a COM object must have its first three compulsory methods, which are QueryInterface, AddRef and Release methods. These methods are subtle. Another major motivation for the formalization of COM is to automate the IUnknown interface. Automating the IUnknown interface substantially reduces
the complexity of the construction of COM implementations and will make COM
easier to use. Several commercial implementations already removed the need to
program IUnknown, but without a solid formal foundation.

The major difference between COM and other standard object models is that
COM does not support implementation inheritance for reuse of existing ob-
jects across components. COM classes cannot inherit implementation from each
other. Reuse is therefore solely based on object composition rather than class
composition. The two forms of object composition supported by COM are called
containment and aggregation. Containment refers to the forwarding of requests
to another object that is solely used (contained) by the forwarding object. Aggre-
gation improves performance in cases where containment would lead to a deep
forwarding chain. Instead of forwarding requests, references to another object’s
interfaces are directly handed out to clients in a way that does not reveal the
forwardee’s identity to the client.

In COM, the use of object composition (based on containment and aggregation)
solely relies on the IUnknown interface. Since COMEL automates the IUnknown
interface, containment and aggregation are supported directly in the COMEL
language. Supporting the constructions of containment and aggregation in the
syntax of COMEL eliminates the complexity of the COM object composition
rules. The operational semantics of COMEL will ensure that the complex aggrega-
tion rules especially are handled correctly.

3 The Approach

Formal semantics is often used to describe the semantics of the programming
languages. There is no uniform approach for formal semantics. Currently, there
are three major approaches to formal semantics: operational semantics, deno-
tational semantics and axiomatic semantics. Operational semantics has been a
favourite approach for most recent research [5, 10] in order to describe the formal
semantics of object-oriented programming languages.

Operational semantics describes the meaning of a programming language by
specifying how programs expressed using it execute on an abstract machine [16].
The evaluation and execution relations are specified by rules in a way directed
by the syntax [11]. Operational semantics itself has two different approaches,
mainly natural semantics and structural operational semantics. Schmidt [13]
discusses the difference between the two approaches. The structural operational
semantics is also known as “small-step” semantics and the natural semantics
is known as “big-step” semantics. The structural operational semantics deals
with describing how the individual steps of the computations take place and the
natural semantics deals with describing how the overall results of executions are
obtained [13].
The COMEL language will use operational semantics in the style of natural semantics [13]. In Drossopoulou et. al. [5], they used small-step semantics to express the Java semantics. Since small-step semantics does not give overall result of evaluation, COMEL will adopt the big-step semantics. To keep the operational semantics very close to the COMEL type rules' format [7], the form for COMEL operational semantics will be almost the same as the form for COMEL type rules. The terminology used will be very close to the type rules terminology used by Cardelli [3]. The following diagram summarizes the terminology used and various auxiliary definitions used to keep the judgments concise. Identifiers in COMEL refer to methods, interfaces, components and variables. Therefore, symbols M, I, O, and x are used to represent Method, Interface, Component, and Variable identifiers respectively. Symbol T and A are used to represent Type and Interface Type respectively.

\[
\begin{array}{|l|l|}
\hline
\Pi ::= \text{identifier} \mapsto \text{value} & \text{state } \Pi \text{ is a partial mapping from identifier to value} \\
\mathcal{V} \in \text{object structures} & \text{identifiers in } \Pi \text{ are generated object identifiers} \\
\Sigma ::= \text{identifier} \mapsto \text{value} & \text{value } V \text{ in } \Pi \text{ is an element of object structures} \\
\text{where } \Sigma \subseteq \text{identifier} \land \Sigma \not\rightarrow \text{value} & \text{state } \Sigma \text{ is a partial mapping from identifier to value} \\
D \in \mathbf{Bool} \cup \mathbf{Int} \cup \mathbf{Comp} & \text{identifiers in } \Sigma \text{ are local variables, self or return} \\
\Sigma' = \Sigma[x \mapsto V] & \text{V in } \Sigma \text{ is either Boolean, Integer or Component} \\
\Sigma' \vdash x \mapsto V & \text{D is a declaration in } \Sigma \text{ where} \\
\text{Comp} \equiv \text{dom}(\Pi) & \text{the binding of variable } x \text{ to value } V \text{ occurs in } \Sigma \\
\Sigma, \Pi \vdash C \rightarrow \Sigma', \Pi' & \text{V in } \Sigma \text{ is either Boolean, Integer or Component} \\
\Sigma, \Pi \vdash C \rightarrow \Sigma', \Pi' & \text{D is a declaration in } \Sigma \text{ where} \\
\Sigma, \Pi \vdash C \rightarrow \Sigma', \Pi' & \text{the binding of variable } x \text{ to value } V \text{ occurs in } \Sigma \\
\Sigma, \Pi \vdash C \rightarrow / \Rightarrow \Sigma', \Pi' & \text{the type of component is equivalent to the dom}(\Pi) \\
\Sigma, \Pi \vdash C \rightarrow / \Rightarrow \Sigma', \Pi' & (\text{domain } \Pi \text{ consists of generated object identifiers}) \\
\Sigma, \Pi \vdash C \rightarrow / \Rightarrow \Sigma', \Pi' & \text{command } C \text{ in } \Sigma, \Pi \text{ will terminate in states } \Sigma', \Pi' \\
x \in \text{dom}(\Sigma) & \text{command } C \text{ in } \Sigma, \Pi \text{ will terminate either} \\
o \in \text{Comp} & \text{normally or non-locally in states } \Sigma', \Pi' \\
x \in \text{dom}(\Sigma) & \text{command } C \text{ in } \Sigma, \Pi \text{ will terminate either} \\
\Sigma, \Pi \vdash E = V \rightarrow \Pi' & \text{normally or non-locally in states } \Sigma', \Pi' \\
x \in \text{dom}(\Sigma) & \text{expression } E \text{ will evaluate to value } V \text{ in states } \Sigma, \Pi \\
o \in \text{Comp} & \text{with side effects leading to state } \Pi' \\
\end{array}
\]

The general form of operational semantics rules used in the following follows the form suggested by Cardelli for type rules [3]. Thus:

\[
\begin{align*}
\Sigma_1 \vdash & \mathcal{S}_1 \rightarrow \Sigma_1' \cdots \Sigma_n \vdash \mathcal{S}_n \rightarrow \Sigma_n' \quad [\text{Conditions}] \\
\Sigma \vdash & \mathcal{S} \rightarrow \Sigma'
\end{align*}
\]

Note that \(\mathcal{S}_1, \ldots, \mathcal{S}_n\) are statements constructed from the immediate constituents of \(\mathcal{S}\). A rule starts with a rule name and follows by a number of \textit{premises} (written above the solid line) and one \textit{conclusion} (written below the solid line). A rule may also have a number of \textit{conditions} (written above the solid line inside
the square brackets) that have to be fulfilled whenever the rule is applied. Rules
with an empty set of premises are called axioms.

Figure 1 summarizes the abstract subset of COMEL syntax described by the
operational semantics. Some operators in the COMEL syntax are left out as
their operational rules correspond closely to their usual operational semantics.
A full COMEL syntax can be found in [6] and its type system can be found
in [7]. Words in italics represent reserved words in the COMEL language.

| Component | ::= COMPONENT O IS
|           |   State₁ ⋯ Stateₙ ExposedInterface₁ ⋯ ExposedInterfaceₙ |
| State     | ::= Contains | Var |
| Contains  | ::= CONTAINS x : InterfaceType = NEW O |
| ExposedInterface | ::= Aggregates | Interface |
| Aggregates | ::= AGGREGATES x₁ |
| Interface | ::= INTERFACE₁ IS Method₁ ⋯ Methodₙ |
| Method    | ::= METHOD M (x₁ : T₁ ⋯ xₙ : Tₙ) : T IS body |
| Var       | ::= VAR x : Type |
| Type      | ::= INTEGER | BOOLEAN | InterfaceType |
| InterfaceType | ::= {I₁ ⋯ Iₙ} |
| body      | ::= Var₁ ⋯ Varₙ Cmd |
| Cmd       | ::= C₁ ; C₂ | x := Expr | RETURN Expr |
|           |           | IF Expr THEN C₁ ELSE C₂ END |
|           |           | WHILE Expr DO Cmd END |
|           |           | CASE x IS InterfaceType : C₁ ELSE C₂ END |
| Expr      | ::= E₁ + E₂ | NEW O | x.I> M(E₁ ⋯ Eₙ) | self.I> M(E₁ ⋯ Eₙ) |
|           |           | E₁ = E₂ | x | Number | NIL | FALSE | TRUE |

**Fig. 1.** The abstract subset of the COMEL syntax
4 The Operational Semantics

This section describes the formal semantics of COMEL. The abstract syntax from Fig. 1 is used to describe the operational semantics.

**Definition 1** If \( o \in \text{Comp} \) then there exists \( o \in \text{dom}(\Pi) \) such that:

\[
\Pi \vdash o \rightarrow \langle O, d', \Sigma' \rangle
\]

where \( d' = \begin{cases} 
\text{d''} & \text{if } o \text{ is a part of an aggregation} \\
\text{o} & \text{if } o \text{ is not a part of an aggregation} 
\end{cases} 
\]

and \( \Sigma' = \langle (V_1 : T_1; \ldots; V_n : T_n) \rangle \)

Definition 1 states that if \( o \) is an object identifier, then there exists object structures in \( \text{dom}(\Pi) \) such that \( o \) is bound to its object structures in state \( \Pi \), where these object structures consist of three fields: the name of its class, \( O \), the component instance of the outer component, \( d' \) (this component instance equals to \( d'' \) if aggregation is used and equal to \( o \) itself if aggregation is not used) and an environment of the component value’s instance variables (this environment consists of values for each instance variables of a certain types).

**Definition 2** If \( V = \langle O, V', \Sigma' \rangle \) and \( \Sigma' = \langle (V_1 : T_1; \ldots; V_n : T_n) \rangle \) then

\[
\begin{align*}
\text{class}(V) &= O; \\
\text{outer}(V) &= V'; \\
\text{fields}(V) &= \Sigma'; \\
\text{field}_i(\Sigma') &= V_1; \ldots; \text{field}_n(\Sigma') &= V_n;
\end{align*}
\]

Definition 2 states that if \( V \) is the value of an element of object structures, then we can access the first object field by class function, the second object field by outer function and the third object field by fields function where \( \text{field}_1 \) to \( \text{field}_n \) are used to access each object’s instance variables.

**Definition 3** If \( x \) is a Component identifier then:

\[
\Pi, \Sigma \vdash x \rightarrow V_x \quad [V_x = \langle O, V', \Sigma' \rangle] \text{ iff } \Pi \vdash \text{and } \Pi \vdash o_x \rightarrow \langle O, V', \Sigma' \rangle
\]

Definition 3 states that if \( x \) is a component identifier, then component identifier evaluates to component value \( V_x \) by 2 evaluations, first component identifier \( x \) evaluates to component instance \( o_x \) and then component instance \( o_x \) is bound to component value \( V \) in \( \Pi \) (a mapping of component instance \( o_x \) to its object identifier \( V \) in \( \text{dom}(\Pi) \)).

As said before all interfaces must inherit from \( IUnknown \) interface which must have \( QueryInterface, AddRef \) and \( Release \) methods. Since COMEL is going to automate the \( IUnknown \) interface and memory management, these three methods are invisible to the implementor. COMEL always makes use of the COM Interface Description Language (COM IDL). This implies that every COMEL component implements COM interfaces that have been described earlier in COM IDL. Therefore, whenever any COM interface is described in IDL as this:

\[
\begin{align*}
\text{class}\langle V \rangle & = O; \\
\text{outer}(V) & = V'; \\
\text{fields}(V) & = \Sigma'; \\
\text{field}_i(\Sigma') & = V_1; \ldots; \text{field}_n(\Sigma') & = V_n;
\end{align*}
\]


import "unknown.idl"; // import IUnknown, IClassFactory
[object, uuid(4411B7FE-EE28-11CE-9054-080036F12502)]
interface IFoo : IUnknown
{
    HRESULT FooMethod(void); }
[object, uuid(4411B7FE-EE28-11CD-9054-080036F12502)]
interface IBar : IUnknown
{
    HRESULT BarMethod(int i); }

the equivalent COMEL component will look like this:

IID IFoo = "4411B7FE-EE28-11CE-9054-080036F12502";
IID IBar = "4411B7FE-EE28-11CD-9054-080036F12502";
COMPONENT FooBar = "4411B7FE-EE28-11CA-9054-080036F12502" IS
    ⟨INTERFACE IUnknown⟩
    ⟨INTERFACE IClassFactory⟩
    INTERFACE IFoo IS
        METHOD FooMethod():HRESULT;
        BEGIN ... END FooMethod;
        END IFoo;
    INTERFACE IBar IS
        METHOD BarMethod(1:INTEGER):HRESULT;
        BEGIN ... END BarMethod;
        END IBar;
    END FooBar.

where ⟨INTERFACE IUnknown⟩ and ⟨INTERFACE IClassFactory⟩ are automatically added and implemented for every COMEL component and are hidden to the COMEL programmer. Note that the unknown.idl contains the IDL descriptions for IUnknown, IClassFactory and all of the COM base types. Therefore, these interfaces and base types are visible to the semantics of the COMEL language.

To automate the IUnknown and IClassFactory interfaces, some functions are introduced to handle various tasks at run-time. They are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int.Jlookup(x, I)</td>
<td>lookup to see whether interface I exists in component identifier x</td>
</tr>
<tr>
<td>met.Jlookup(IID J, M)</td>
<td>lookup to get the commands for specified method M for interface J</td>
</tr>
<tr>
<td>class_fact(O)</td>
<td>get the class factory of a component O</td>
</tr>
<tr>
<td>creat.Inst(c)</td>
<td>create a component instance of a class factory c</td>
</tr>
</tbody>
</table>

The interface IUnknown consists of the QueryInterface, AddRef and Release methods. To implement a QueryInterface method in COMEL, a function int.Jlookup(x, I) is used at run-time, where this function will do a lookup in a table to see whether interface I is implemented by the component x. In COMEL syntax this is equivalent to x.I.
Definition 4 An interface lookup is equivalent to $x.I$ where:

$$
x.I \triangleq x.\text{QueryInterface}(IID_I, &\text{ip})$$  [COM]
$$= (x, \text{Int-table})$$

$$x.I \triangleq \text{int.lookup}(x, I)$$  [COMEL]
$$= (\text{self, int.lookup}(x, IID_I))$$

Definition 4 states that whenever $x.I$ is used, this means that an interface lookup function is required to look up the interface $I$ within the component $x$. From the COM side, interface lookup $x.I$ is by definition equivalent to, $\triangleq$, calling the QueryInterface() function where this query will return the interface table (Int-table) and interface pointer $\text{ip}$ of requested interface via its out parameter. Then this interface pointer can be used to ask for a method that exist in the specified interface. From the COMEL side, interface lookup $x.I$ is by definition equivalent to, $\triangleq$, calling the $\text{int.lookup}()$ function at run-time ($\text{int.lookup}()$ will be specified formally further below), where this lookup will return method table (Met-table) of the requested interface, if interface $I$ exists; otherwise $\text{int.lookup}()$ returns $\text{NIL}$. Then this method table (method table consists of mapping from method names to methods’ commands) can be used by the $\text{met.lookup}$ function where $\text{met.lookup}$ function will return the commands that belong to the specified method.

The AddRef and Release methods in the IUnknown interface are automatically added to a method’s statements such as assignment, a method call, creating a new instance of another component and using a local variable as parameter. A different operational semantics for these statements are needed in order to capture the binding of these statements for the memory management (not presented here).

The interface IClassFactory consists of the CreateInstance and LockServer methods. To implement a CreateInstance method in COMEL, a function $\text{createInst}(c)$ is used at run-time, where this function creates a component instance $o_i$ by calling on a class factory object $c$.

Definition 5 Creating a component instance is equivalent to NEW $O$ where:

$$\text{NEW } O \triangleq \text{CreateInst}(\text{CLSID}_O, \cdots, IID_{IUnknown}, &\text{ipc})$$  [COM]
$$\text{NEW } O \triangleq create\_inst(\text{classfact}(\text{CLSID}_O))$$  [COMEL]
$$= o_i$$

Definition 5 states that whenever NEW $O$ is used, this is actually asking for the creation of a new component instance. From the COM side, creating a component instance is by definition equivalent to calling the CreateInstance() function, where this call will return the component instance via the IUnknown interface pointer $\text{ipc}$ of requested component $O$. From the COMEL side, creating a component instance is by definition equivalent to first calling the classfact()
function at run-time, where this will load (if required) and get the class
factory $c$ of the specified component $O$, then this class factory $c$ is passed to the
\texttt{crea\_inst()} function to request a component instance $o_i$ from the class
factory $c$. The \texttt{crea\_inst(c)} function is equivalent to the \texttt{CreateInstance}
method in the IClassFactory interface and the \texttt{class\_fact(CLSID,$O$)} function is equiva-
lent to the \texttt{CoGetClassObject} method in the COM library with extra task that
it will also load the class factory if required. In COM, \texttt{CreateInstance} is a
wrapper API function that does the sequence of calls \texttt{CoGetClassObject}, \texttt{IClass-
Factory::CreateInstance} and \texttt{IClassFactory::Release}. Thus, COMEL automates
this sequence of calls to \texttt{class\_fact}, \texttt{crea\_inst} and \texttt{release} where \texttt{release} function
will be added automatically. The formal specification of the \texttt{release} function is
not discussed in this paper. The \texttt{crea\_inst()} function will return a component
instance $o_i$. In addition to this component instance $o_i$, COMEL’s typecase state-
ment can be used in order to query or obtain additional interface pointers.

The LockServer method in IClassFactory is not used directly in COMEL but
can be converted to COMEL syntax if necessary. So far, the mechanism of how
to connect and activate the server has not been discussed. Currently, LockServer
is not modelled in COMEL. For simplicity, COMEL assumes that all interfaces
reside only on in-process server. Therefore, loading the appropriate class factory
is handle by \texttt{class\_fact()} function.

COMEL does not fully formalize the COM environment. However, COMEL
relies on the COM environment and thus formalizes a subset of the COM en-
vIRONMENT, in particular, the type libraries. A COM type library is a collection
of type information elements for one or more objects and one or more interfaces
as well as a repository for type definitions and module export lists [9]. The type
information describes an object’s Class Identifier (CLSID), its interfaces, their
Interface Identifiers (IIDs), their member functions, and the return value and
arguments of those functions. The COM type library and type information are
being formalized in COMEL.

**Definition 6** The COM type library is:

$$\Gamma_{CLSID} : CLSID \rightarrow \wp(IID)$$

where

- $\Gamma_{CLSID}$ is maintained by COM’s CLSID repository
- and maps CLSID to set of interfaces
  - (however, all components are statically modelled to have type \{ \})

and

$$\Gamma_{IID} : IID \rightarrow \wp(MT)$$

where

- $\Gamma_{IID}$ is maintained by COM’s IID repository
- and maps IID to set of method types

and

$$MT = T_1 \cdots T_n \rightarrow T$$

where
A method type consists of argument types $T_1 \cdots T_n$, and a result type $T$
\[\text{Dom}(MT) = T_1 \cdots T_n\]
\[\text{Range}(MT) = T\]

Definition 6 is part of the definition introduced for the type rules of the COMEL language [7]. The environment $\Gamma_{\text{CLSID}}$ and $\Gamma_{\text{IID}}$ are used for the type rules and are visible to the semantics of the COMEL language. Thus, by having the CLSID of a component, environment $\Gamma_{\text{CLSID}}$ can be used to get the interfaces that the component supported and by having the IID of an interface, environment $\Gamma_{\text{IID}}$ can be used to get the method types $T_1 \cdots T_n \rightarrow T$ (ie method argument types and result type) that the interface implemented.

In COM, the use of object composition which is based on the containment and aggregation methods will solely rely on the IUnknown interface. Since COMEL automates the IUnknown interface, there should be a mechanism in order to remember the outer IUnknown interface pointer. This is important in order for aggregation to work properly. To implement the aggregation rule in COMEL, a function $\text{intLookup}(x, I)$ is used at run-time, where this function will do an appropriate calling depending on whether aggregation is used or not.

**Definition 7** If $x$ is a Component identifier, and $V = \langle O \ V' \ \Sigma' \rangle$, then for any given $x, I \triangleq \text{intLookup}(x, I)$:

\[
\begin{align*}
\Pi, \Sigma \vdash x = V & \quad [V = \langle O \ V' \ \Sigma' \rangle] \\
\Pi \vdash \text{outer}(V) = V' & \quad [V' = V'] \\
\Gamma_{\text{CLSID}} \vdash O : \{I\} & \\
\Pi, \Sigma \vdash \text{intLookup}(x, I) = V & \quad \Pi, \Sigma \vdash x = V = \langle O \ V' \ \Sigma' \rangle \\
\Pi \vdash \text{outer}(V) = V'' & \quad [V' \neq V''] \\
\Gamma_{\text{CLSID}} \vdash O. I \text{ aggregates from some } x' & \\
\Pi, \Sigma \vdash \text{intLookup}(x', I) = V'' & \\
\Pi, \Sigma \vdash \text{intLookup}(x, I) = V''
\end{align*}
\]

Definition 7 states that if $x$ is a Component identifier and $V$ is the value of an object identifier, then $\text{intLookup}(x, I)$ is by definition equivalent to evaluating $x$ to object value $V$ in states $\Pi$ and $\Sigma$ first (from Definition 3). Then, if aggregation is not used, the second field of component value $V$, outer$(V)$, will equal to $V$. Therefore, the class of component value $V$, $O$, will support the interface $I$ in $\Gamma_{\text{CLSID}}$. If aggregation is used, the second field of component value $V$, outer$(V)$, will not equal to $V$. Therefore, the class of component value $V$, $O$, will indicate that it is aggregated from other component $x'$ in $\Gamma_{\text{CLSID}}$. Then the interface is accessed recursively via $\text{intLookup}()$ function.

However, if the component instance is not present, a component self is used instead of a variable $x$ to indicate that it is called within the component itself.

Once $\text{intLookup}()$ returns the method table, this method table can be used to get the commands of the method via the $\text{metLookup}()$ function.

**Definition 8** If $\text{intLookup}(x, I) = V$ then $\text{metLookup}(\text{IID}_I, M)$ is:

\[\Pi \vdash \text{metLookup}(\text{IID}_I, M) = C \quad \langle\langle x_1 \cdots x_n\rangle\rangle\]
4.1 Declarations

For declarations in COMEL, the Contains declaration needs to have the semantics rule because of the creation of new instance is used. The constant and variable declarations are the other declarations that need to have the semantics rule.

Figure 2 describes the evaluation of declarations. In a Contains declaration, the creation of a new instance of component O is evaluated first; if it succeeds, it will bind the instance of component o_i to the value V (where V consists of the component class name O, the value of the outer component V_outer (this value is equal to o_outer) and the initial values for instance variables V_1 \cdots V_n of types T_1 \cdots T_n). The instance created is bound to the variable used in the CONTAINS statement. For Variable declarations, FALSE is the initial value of BOOLEAN variables, 0 is the initial value of INTEGER variables and NIL is the initial value of set of interfaces variables.

\[
\begin{align*}
(\text{Decl Contains}) & \\
\Sigma \vdash \text{outer} = o_i \\
\{o_i \notin \text{dom}(\Sigma), \text{create}_\text{inst}(\text{class}_\text{fact}(\text{CLSID}_O)) = o_i, \\
V = \{O \ V_{\text{outer}} V_1 : T_1; \cdots; V_n : T_n\}, V_i \text{ is the initial value for } T_i, i = 1 \cdots n\} \\
\Sigma \vdash \text{CONTAINS} x : \{I_1, \cdots, I_n\} = \text{NEW} O \rightarrow \Sigma [o_i \mapsto V, x \mapsto o_i]
\end{align*}
\]

(Decl VartoTypeBool)

\[
\Sigma \vdash \text{VAR} x : \text{BOOLEAN} \rightarrow \Sigma[x \mapsto \text{FALSE}]
\]

(Decl VartoTypeInt)

\[
\Sigma \vdash \text{VAR} x : \text{INTEGER} \rightarrow \Sigma[x \mapsto 0]
\]

(Decl VartoTypeComp)

\[
\Sigma \vdash \text{VAR} x : \{\} \rightarrow \Sigma[x \mapsto \text{NIL}]
\]

Fig. 2. Declaration evaluation

4.2 Commands

Since general commands and expressions in COMEL can cause side effects, the rule form \(\Pi, \Sigma \vdash \exists V : \Pi'\), \(\Sigma'\) will be used. This will allow the execution of statement \(\exists\) to be evaluated to value \(V\) from states \(\Pi, \Sigma\) and at the same time leading to the potentially different states \(\Pi', \Sigma'\) because of side effects. Whenever
the notation \( \rightarrow / \Rightarrow \) is used, this means that the statement terminates either normally \( \rightarrow \) or non-locally \( \Rightarrow \) (non-local termination is caused by an interruption of the evaluation, in particular, the return command in a command sequence). Whenever \( \rightarrow / \Rightarrow \) is used both in premises and conclusion of a rule, if the premises terminate normally then the conclusion of the rule must also terminate normally and if the premises terminate non-locally then the conclusion of the rule must also terminate non-locally.

Figure 3 describes the evaluation of method body and command sequence. In a method body, which consists of a list of variable declarations followed by a list of commands, variable declarations will be evaluated first, then the command sequence is evaluated where the local variables used are bound to its values in command environment. For a command sequence, the evaluation is from left to right. However, if the command sequence is interrupted, say by the return command, all further commands in sequential composition will be ignored and an interrupted evaluation will cause a non-local termination. An empty command sequence has no effect on the states.

\[
\begin{align*}
(Pm) & \quad \Pi, \Sigma \vdash D_j \rightarrow \Pi''', \Sigma''' = \Sigma[x_j \mapsto V_j] \quad j = 1 \ldots n \quad \text{(local variable } x_j) \\
(Ps) & \quad \Pi, \Sigma[x_1 \mapsto V_1 \ldots x_n \mapsto V_n] \vdash C_1 \ldots C_n \rightarrow \Pi', \Sigma' \quad \text{[}V_1 \ldots V_n \text{ are initial values]} \\
(Pc) & \quad \Pi, \Sigma \vdash D_1 \ldots D_n, C_1 \ldots C_n \rightarrow \Pi', \Sigma'[x_1 \mapsto V_1 \ldots x_n \mapsto V_n].
\end{align*}
\]

\[
\begin{align*}
(Ps) & \quad \Pi, \Sigma \vdash C_1 \rightarrow \Pi''', \Sigma''' \quad \Pi'', \Sigma''' \vdash C_2 \rightarrow / \Rightarrow \Pi', \Sigma' \\
(Pc) & \quad \Pi, \Sigma \vdash C_1; C_2 \rightarrow / \Rightarrow \Pi', \Sigma'
\end{align*}
\]

\[
\begin{align*}
(Ps) & \quad \Pi, \Sigma \vdash C_1 \Rightarrow \Pi', \Sigma' \\
(Pc) & \quad \Pi, \Sigma \vdash \epsilon \Rightarrow \Pi', \Sigma'
\end{align*}
\]

\[
\begin{align*}
(Pm) & \quad \Pi, \Sigma \vdash D_j \rightarrow / \Rightarrow \Pi''', \Sigma'''
\end{align*}
\]

Fig. 3. Body and sequence commands evaluation

Figure 4 describes the evaluation of assignment, return, if, while and typecase commands. In an assignment command, the expression on the right-hand side of the assignment is evaluated to a value and at the same time, because of potential side effects, this will also evaluated to a new state. The variable used on the left-hand side of the assignment will then be bound to this value in the new state. For a return command, the expression in the return command is evaluated first, and at the same time, because of potential side effects, this will also evaluate to a new state. A return statement causes a non-local termination in the new state and the return variable is bound to the value that the return command’s evaluated to.
(Cmd Assign)
\[\Pi, \Sigma \vdash E = V \rightarrow \Pi', \Sigma'\]
\[\Pi, \Sigma \vdash x := E \rightarrow \Pi', \Sigma'[x \mapsto V]\]

(Cmd Return)
\[\Pi, \Sigma \vdash E = V \rightarrow \Pi', \Sigma'\]
\[\Pi, \Sigma \vdash \text{RETURN } E \Rightarrow \Pi'', \Sigma'[\text{return } \mapsto V]\]

(Cmd If True)
\[\Pi, \Sigma \vdash E = \text{TRUE} \rightarrow \Pi', \Sigma' \quad \Pi', \Sigma' \vdash C_1 \rightarrow / \Rightarrow \Pi'', \Sigma''\]
\[\Pi, \Sigma \vdash \text{IF } E \text{ THEN } C_1 \text{ ELSE } C_2 \text{ END } \rightarrow / \Rightarrow \Pi'', \Sigma''\]

(Cmd If False)
\[\Pi, \Sigma \vdash E = \text{FALSE} \rightarrow \Pi', \Sigma' \quad \Pi', \Sigma' \vdash C_2 \rightarrow / \Rightarrow \Pi'', \Sigma''\]
\[\Pi, \Sigma \vdash \text{IF } E \text{ THEN } C_1 \text{ ELSE } C_2 \text{ END } \rightarrow / \Rightarrow \Pi'', \Sigma''\]

(Cmd While True)
\[\Pi, \Sigma \vdash E = \text{TRUE} \rightarrow \Pi'', \Sigma'' \quad \Pi'', \Sigma'' \vdash C \rightarrow \Pi''', \Sigma'''\]
\[\Pi, \Sigma \vdash \text{WHILE } E \text{ DO } C \text{ END } \rightarrow / \Rightarrow \Pi', \Sigma'\]

(Cmd While False)
\[\Pi, \Sigma \vdash E = \text{FALSE} \rightarrow \Pi', \Sigma'\]
\[\Pi, \Sigma \vdash \text{WHILE } E \text{ DO } C \text{ END } \rightarrow \Pi', \Sigma'\]

(Cmd While TrueWithReturn)
\[\Pi, \Sigma \vdash E = \text{TRUE} \rightarrow \Pi', \Sigma' \quad \Pi', \Sigma' \vdash C \rightarrow \Pi''', \Sigma'''\]
\[\Pi, \Sigma \vdash \text{WHILE } E \text{ DO } C \text{ END } \Rightarrow \Pi'', \Sigma''\]

(Cmd TypeCase True)
\[\Pi, \Sigma \vdash x = V \quad [V = \{ O. V' \Sigma' \}]\]
\[\Sigma \vdash x \text{ IS } \{I_1 \ldots I_n\} = \text{TRUE} \quad [\text{CLS} \vdash O : \{I_1 \ldots I_n\}]\]
\[\Pi, \Sigma \vdash C_1 \rightarrow / \Rightarrow \Pi''', \Sigma'''\]
\[\Pi, \Sigma \vdash \text{CASE } x \text{ IS } \{I_1 \ldots I_n\} : C_1 \text{ ELSE } C_2 \text{ END } \rightarrow / \Rightarrow \Pi'', \Sigma''\]

(Cmd TypeCase False)
\[\Pi, \Sigma \vdash x = V \quad [V = \{ O. V' \Sigma' \}]\]
\[\Sigma \vdash x \text{ IS } \{I_1 \ldots I_n\} = \text{FALSE} \quad [\text{CLS} \not\vdash O : \{I_1 \ldots I_n\}]\]
\[\Pi, \Sigma \vdash C_2 \rightarrow / \Rightarrow \Pi''', \Sigma'''\]
\[\Pi, \Sigma \vdash \text{CASE } x \text{ IS } \{I_1 \ldots I_n\} : C_1 \text{ ELSE } C_2 \text{ END } \rightarrow / \Rightarrow \Pi'', \Sigma''\]

(Cmd TypeCase NIL)
\[\Pi, \Sigma \vdash x = \text{NIL}\]
\[\Sigma \vdash x \text{ IS } \{I_1 \ldots I_n\} = \text{FALSE} \quad \Pi, \Sigma \vdash C_2 \rightarrow / \Rightarrow \Pi', \Sigma'\]
\[\Pi, \Sigma \vdash \text{CASE } x \text{ IS } \{I_1 \ldots I_n\} : C_1 \text{ ELSE } C_2 \text{ END } \rightarrow / \Rightarrow \Pi', \Sigma'\]

Fig. 4. Assignment, return, if, while and typecase commands evaluation
In an if command in Fig. 4, the condition is evaluated first and because of potential side effects, this will also evaluate to a new state. If the condition evaluates to *TRUE*, then the first branch is executed; otherwise the second branch is executed. For a while command, the condition is evaluated first; if it evaluates to *TRUE*, the command will be evaluated followed by another iteration of the while loop, otherwise the evaluation will terminate and the state is left where the expression evaluation took it. However, if the evaluation of the command is interrupted, say by the return command, all further commands in the while loop will be ignored and an interrupted evaluation will cause a non-local termination.

In a type-case command in Fig. 4, the component identifier is evaluated first. If this component identifier equals to a value V, then its class supports interfaces \( I_1 \cdots I_n \) in \( I_{\text{CLSID}} \). If however in \( I_{\text{CLSID}} \), the class of the component identifier does not derive to support interfaces \( I_1 \cdots I_n \), then the component identifier that supports the required interfaces will evaluate to *FALSE*. If the component identifier that supports the required interfaces evaluates to *TRUE*, the first branch is executed, otherwise the second branch is executed. However, if the evaluation of the component identifier is equal to *NIL*, the component identifier that supports the required interfaces will evaluate to *FALSE* and the second branch is executed.

![Fig. 5. Addition, condition and new expressions evaluation](image)

### 4.3 Expressions

Most expressions in COMEL can have side effects. Therefore the evaluation of each expression will equal to a value from one state and at the same time leading
(Expr MethodCall)
\[ \Sigma, \Pi \vdash x = V \rightarrow \Pi_0 \quad [V = \{ O V' \Sigma' \}] \]
\[ \Sigma, \Pi_j \vdash E_j = V_j \rightarrow \Pi_{j+1} \quad (j = 0 \cdots n - 1) \]
\[ \Pi_n \vdash \text{int	extunderscore lookup}(x, I) = V'' \]
\[ \Pi_n \vdash \text{met	extunderscore lookup}(\text{HD}, I, M) = C \langle \langle x_1, \cdots, x_n \rangle \rangle \]
\[ x_1 = V_1, \cdots, x_n = V_n, \text{self} = V'', \Pi_n \vdash C \rightarrow \Sigma''', \Pi' \]
\[ \Sigma''' \vdash \text{return} = V''' \]
\[ \Sigma, \Pi \vdash x. I \rightarrow M(E_1 \cdots E_n) = V''' \rightarrow \Pi' \]

(Expr SelfMethodCall)
\[ \Sigma, \Pi \vdash \text{self} = V \rightarrow \Pi_0 \quad [V = \{ O V' \Sigma' \}] \]
\[ \Sigma, \Pi_j \vdash E_j = V_j \rightarrow \Pi_{j+1} \quad (j = 0 \cdots n - 1) \]
\[ \Pi_n \vdash \text{int	extunderscore lookup}(\text{self}, I) = V'' \]
\[ \Pi_n \vdash \text{met	extunderscore lookup}(\text{HD}, I, M) = C \langle \langle x_1, \cdots, x_n \rangle \rangle \]
\[ x_1 = V_1, \cdots, x_n = V_n, \text{self} = V'', \Pi_n \vdash C \rightarrow \Sigma''', \Pi' \]
\[ \Sigma''' \vdash \text{return} = V''' \]
\[ \Sigma, \Pi \vdash \text{self}. I \rightarrow M(E_1 \cdots E_n) = V''' \rightarrow \Pi' \]

Fig. 6. Method Call evaluation

(Expr MethodLocalVar)
\[ \Sigma \vdash x \rightarrow V \]
\[ \Sigma, \Pi \vdash x = V \rightarrow \Pi \]

(Expr CompLocalVar)
\[ \Sigma \vdash \text{self} \rightarrow V_{\text{self}} \quad [x \notin \text{dom}(\Sigma)] \]
\[ \Pi \vdash V_{\text{self}} \rightarrow \{ O V' \Sigma' \} \]
\[ \Sigma' \vdash x \rightarrow V \]
\[ \Sigma, \Pi \vdash x = V \rightarrow \Pi \]

Fig. 7. Identifier evaluation

(Expr Number)
\[ \Sigma \vdash \text{Int} = \nu \]

(Expr Null)
\[ \Sigma \vdash \text{NIL} = \{ \} \]

(Expr False)

(Expr True)
\[ \Sigma \vdash \text{FALSE} = \text{FALSE} \quad \Sigma \vdash \text{TRUE} = \text{TRUE} \]

Fig. 8. COMEL’s axioms
to the potential different state, because of side effects. Whenever V is used, this
implies that the actual machine operator is used. If E is used to replace V, this
means that the operator is the reserved operator in COMEL language. Figure 5
describes the evaluation of addition, equal and new expressions. In \( E_1 + E_2 \)
expression, where + is a reserved operator, the expression \( E_1 \) is evaluated to a
value \( V_1 \) and at the same time, because of potential side effects, this will also
evaluate to a new state. Then, the expression \( E_2 \) is evaluated to a value \( V_2 \)
and at the same time this will evaluate to a new state because of potential side
effects. Finally, \( V_1 + V_2 \) is evaluated, where + is a machine operator. Again the
evaluation of \( V_1 + V_2 \) will terminate to a new state because of potential side
effects.

In \( E_1 = E_2 \) expression in Fig. 5, where = is a reserved operator, the expression
\( E_1 \) is evaluated to a value \( V_1 \) and at the same time, because of potential side
effects, this will also evaluate to a new state. Then, the expression \( E_2 \) is evaluated
to a value \( V_2 \) and at the same time this will evaluate to a new state because of
potential side effects. Finally, \( V_1 = V_2 \) is evaluated, where = is a machine
operator. The evaluation will then evaluate to \( TRUE \) if it satisfied the condition,
otherwise it evaluated to \( FALSE \). Again the evaluation of \( V_1 = V_2 \) will terminate
to a new state because of potential side effects.

In a new expression in Fig. 5, the creation of a new instance of component O
is evaluated; if it is succeed, it will bind the instance of component \( o_i \) to the
value \( V \), where \( V \) consists of the component class name \( O \), the value of the outer
compound \( V_{outer} \) (this value is equal to \( NIL \)) and the initial values for instance
variables \( V_1 \cdots V_n \) of types \( T_1 \cdots T_n \). The \textit{outer} variable is bound to the value
that the new expression's evaluated to.

Figure 6 describes the evaluation of a method call. For a method call expression,
the instance of component is evaluated first. This component instance is equal
to a value \( V \) (where \( V \) consists of the component class name \( O \), the value of the
outer component \( V' \) (this value is equal to \( NIL \) if aggregation is not used) and the
environment of the component value's instance variables). Then, the argument
expressions are evaluated from left to right. The \textit{int\_lookup()} and \textit{met\_lookup()} functions
are used to get the command for the specified method and then the
command is evaluated. Evaluation of a method call will terminate to value \( V''' \),
which will be the same type of method declaration, and at the same time will
evaluate to a new state \( J'' \) because of potential side effects. If the component is
missing, a method call is equivalent to a call of a \textit{self} interface method within
the component.

Figure 7 evaluates an expression being a single identifier and Figure 8 lists the
axioms of the COMEL language. Number, Null, False and True expressions are
the axioms for \textit{INTEGER, NIL, FALSE and TRUE} respectively.
5 Subject Reduction Theorem and Soundness

The consistency of the COMEL operational semantics with its typing rules is established by a subject reduction theorem [1]. The subject reduction theorem [15] states that if an expression $E$ has type $T$ in type environment $\Gamma$, if state $\Sigma$ respects environment $\Gamma$, ($\models \Sigma : \Gamma$), and if $E$ evaluates to a value $V$ in state $\Sigma$ with side effects leading to state $\Sigma'$, then $V$ possesses type $T'$ in state $\Sigma'$, where type $T'$ in COMEL language is either type INTEGER, BOOLEAN or set of interfaces, and $T'$ is compatible with $T$.

Theorem 1 (Subject Reduction) If $\Gamma \vdash E : T$, $\models \Sigma : \Gamma$ and $\Sigma \vdash E = V \to \Sigma'$ then:

$$\Sigma' \models V : T' \text{ and } T' \prec_w T$$

where

$$T' \prec_w T = \begin{cases} 
T' = T \text{ iff } T = \text{INTEGER} \\
T' = T \text{ iff } T = \text{BOOLEAN} \\
T' \supseteq T \text{ iff } T' = \{ I_1 \cdots I_n \} \text{ and } T \text{ is a set of interfaces type }
\end{cases}$$

Based on the subject reduction theorem and arguments that well-typed programs produce well-typed answers, we can deduce the soundness theorem. The type soundness theorem [15] states that if an expression $E$ has type $T$ in type environment $\Gamma$ and expression $E$ is evaluated by successive reductions into a new state ($E_1 \to E_2 \to \cdots$), then the reduction may either continue forever ($E \uparrow$ where $E \uparrow \equiv E_1 \to E_2 \to \cdots$), or reach a final state where no further evaluation is possible ($\Sigma \vdash E \to V$ and $\Sigma \vdash E_k \to E_{k+1}$).

Theorem 2 (Soundness) If $\Gamma \vdash E : T$, $\models \Sigma : \Gamma$ and $E_1 \to E_2 \to \cdots$ then:

either $E \uparrow \equiv E_1 \to E_2 \to \cdots \infty$
or $\Sigma \vdash E \to V$, $\models V : T'$ and $T' \prec_w T$

where

$$T' \prec_w T = \begin{cases} 
T' = T \text{ iff } T = \text{INTEGER} \\
T' = T \text{ iff } T = \text{BOOLEAN} \\
T' \supseteq T \text{ iff } T' = \{ I_1 \cdots I_n \} \text{ and } T \text{ is a set of interfaces type }
\end{cases}$$

The proof of the subject reduction theorem and soundness theorem are still under major construction. Our work is still progressing on these issues. The intention is to used the theorem prover to assist the proof. The theorem prover is also needed in order to proof the soundness of the COMEL type system with the operational semantics.

6 Future Directions of the Work and Conclusions

One major purpose of developing a formal model for COM is the hope that it can be later generalized to a formal model for independently extensible systems. Work is still progressing on the soundness and memory management of COMEL.
Formalization of COM is needed in order to understand the COM's informal and complex rules better. Due to the very large number of COM concepts and rules, some of the COM rules, which are more complex and not easy to formalize, are being omitted. These include security, licensing, marshaling, threading models and remote objects.

Acknowledgements. We would like to thank Chris Ho-Stuart and Paul Roe for their ideas and comments towards improving the paper.

References

Modelling Specification-based Testing of Interactive Systems

Ian MacColl and David Carrington

Software Verification Research Centre
Department of Computer Science & Electrical Engineering
The University of Queensland
{iann,davec}@csee.uq.edu.au

Abstract. In this paper we present an abstract model of specification-based testing of interactive systems. Interactive systems can be analysed and developed in terms of functionality, presentation and behaviour. Testing information for interactive systems can be derived from formal specifications of each of these aspects. The model incorporates all three aspects. We plan to use the model to improve a framework we have developed for specification-based testing of interactive systems.

1 Introduction

An interactive system is one in which “each user entry causes a response from or action by the system” [11]. Interaction is a significant component of contemporary software systems and graphical user interfaces are now almost universal. Interactive systems can be viewed as conceptually and architecturally composed of functionality, presentation and interaction components [24].

Ostrand et al. [19] note that correctness of an interactive system involves both proper behaviour of the user interface (interaction) and proper computations of the underlying application (functionality). Testing is one method that is used to increase confidence in software correctness, but testing user interface software is difficult. Although testing of user interface software has elements in common with other software testing, it also presents a number of challenges: size and complexity (compared to functionality); graphical and other presentations; multiple, asynchronous input devices; (apparently) modeless operation; and rapid semantic feedback. Interactive systems have much in common with reactive systems but differ in having a human user to whom information is presented.

Solutions proposed to meet these challenges mainly involve automating test execution with tools providing: widget-based verification in addition to bitmap-based verification; scripting, especially with encapsulation of dependencies; automatic synchronisation; and separate data, control and usability testing [10, 12, 13]. Apart from abstracting away from the concrete details of graphical presentations and input events, little guidance is provided on what tests to perform or what results to expect. Formal specifications can be used to provide this guidance.
We are investigating black-box testing of interactive systems based on abstract, multiparadigm specifications. In previous papers, we derive testing information from such a specification to confirm that each view provides useful, but different, testing information [16], and we propose an initial framework for structuring the testing information derived from each of the views [17].

The framework is an extension of the Test Template Framework (TTF) [23], a formal, abstract model of testing, used to derive a hierarchy of test information. The TTF is designed for model-based notations, such as Z, and our extension accommodates behaviour-based notations such as CSP.

In this paper we present an abstract model of specification-based testing of interactive systems. In the next section we present a model of interactive systems which incorporates model-based and behaviour-based notations. In Sections 3 and 4 we present models of testing derived from model-based and behaviour-based notations respectively. We relate the Test Template Framework to each model. In Section 5 we discuss related work and conclude. We assume basic familiarity with Z and CSP.

2 Interactive Systems

We introduce a model of interactive systems based on models presented by Duke and Harrison [3, 4]. The model distinguishes three views of the system, functionality, presentation and interaction, consistent with the Seeheim model [8], Arch/Slinky metamodel [24], and MVC [14] and PAC [1] architectures.

Duke and Harrison demonstrate that presentation can be seen as an abstraction of functionality, and that both can be specified in a model-based notation, such as Z. We factor out information common to both views and define a generic, model-based specification $M$ that can be instantiated to define a functionality or presentation view. The generic parameters represent events ($E$) and state ($S$). The model defines a set of states the view can occupy ($states$), an initial set of states ($initial$) and a set of actions corresponding to events that trigger state transitions ($actions$). The predicate requires that the initial states be in $states$ and that transitions are only possible between elements of $states$.

$$M[E, S]$$

$states : P S$
$initial : P S$
$actions : E \rightarrow (S \leftrightarrow S)$
$initial \subseteq states$
$\forall e : dom actions \cdot actions e \subseteq states \times states$

We define a generic behaviour-based specification ($B$) that consists of a set of event sequences ($traces$) which is prefix-closed. $B$ can be instantiated to provide an interaction view.

---

1 We use the term view to refer to a partial specification.
\[
B[E] \\
\text{traces} : P(\text{seq } E) \\
\forall s, t : \text{seq } E \cdot s \cdot t \in \text{traces} \Rightarrow s \in \text{traces}
\]

An interactive system (\textit{System}) is modeled with generic parameters representing events \(E\), states \(S\) and percepts \(P\). Percepts model perceivable aspects of the system presentation. We instantiate functionality \(FS\), presentation \(PS\) and interaction \(IS\) views with additional structures to relate traces to states \(result\), traces to percepts \(view\) and states to percepts \(render\). The schema predicate maintains consistency of the sets of event sequences and defines the three additional relations.\(^2\)

\[
\text{System}[E, S, P] \\
\text{FS} : M[E, S] \\
\text{PS} : M[E, P] \\
\text{IS} : B[E] \\
\text{result} : \text{seq } E \leftrightarrow S \\
\text{view} : \text{seq } E \leftrightarrow P \\
\text{render} : S \leftrightarrow P
\]

\[
\text{dom FS.actions} = \text{dom PS.actions} \\
\forall t : \text{IS.traces} \cdot \text{ran } t \subseteq \text{dom FS.actions} \\
\text{dom result} = \text{dom view} = \text{IS.traces} \\
\forall t : \text{IS.traces}; s : S \cdot t \text{ result } s \Leftrightarrow s \in (\langle t, s \rangle \text{FS.actions}) \{FS.initial\} \\
\forall t : \text{IS.traces}; p : P \cdot t \text{ view } p \Leftrightarrow p \in (\langle t, s \rangle \text{PS.actions}) \{PS.initial\} \\
\forall s : S; p : P \cdot s \text{ render } p \Leftrightarrow (\exists t : \text{IS.traces} \cdot t \text{ result } s \wedge t \text{ view } p)
\]

In this section we have defined a model of interactive systems. In the next section we introduce a model of specification-based testing of the functionality and presentation views of such systems.

\(^2\) We require a supporting definition of distributed composition to define \text{result} and \text{view}:

\[
[X] \\
\langle \rangle : \text{seq } (X \leftrightarrow X) \rightarrow (X \leftrightarrow X) \\
\langle \rangle = \text{id } X \\
\forall r : X \leftrightarrow X; s : \text{seq } (X \leftrightarrow X) \cdot \\
\begin{align*}
\langle \langle r \rangle \wedge s \rangle &= r \langle s \rangle 
\end{align*}
\]
3 Model-based SBT

Specification-based testing is concerned with the use of a specification to guide testing (in contrast to use of the implementation). We are interested in using formal specifications for this purpose. In Section 3.1 we introduce a model of model-based specification-based testing and in Section 3.2 we apply it to the Test Template Framework.

3.1 Model

In a seminal work on using Z specifications to support testing, Hall [9] outlines a formalisation of testing, originally due to Goodenough and Gerhart [7]. The theory is based on a view of programs as functions. Specifications and programs are defined as mappings from input to output.

\[
\begin{align*}
Spec[S] & := S \rightarrow S \\
Prog[S] & := S \rightarrow S
\end{align*}
\]

The basic idea is that there exists some relation between a specification and a program such that the program is an implementation of the specification. This is similar to refinement, retrieval, abstraction or representation relations, and here is called OK.

\[
\begin{array}{c}
[S]
\
\text{OK : Spec}[S] \leftrightarrow \text{Prog}[S]
\end{array}
\]

An aim of testing is to increase confidence in OK (or, conversely, to find where it is flawed). For a specification \( s \), this is achieved by selecting elements in the domain of \( s \), mapping these abstract elements to concrete elements, executing a candidate implementation \( p \) with each concrete element as input and evaluating the resulting output. A specification can be used for both selection and evaluation. Test execution can be modeled by an implementation relation \( \text{imp} \) between a specification and a program.

\[
\begin{array}{c}
[S]
\
\text{imp : Spec}[S] \leftrightarrow \text{Prog}[S]
\end{array}
\]

\[
\forall s : Spec[S]; p : Prg[S] \bullet \exists \text{ imp } p \iff
\forall t : \text{ dom } s \bullet \exists f : S \leftrightarrow S \bullet
(s \circ f)(t) = (f \circ p)(t)
\]

Exhaustive testing involves executing program \( p \) with every possible input (i.e., every element of the domain of \( s \)) which is generally impractical. Test selection involves identifying subsets of inputs that are expected to produce similar results. Selection can be based on either a specification (black-box testing) or a program (white-box testing) and is modeled by a testing method.

\[
\text{method}[S] := \text{Spec}[S] \times \text{Prog}[S] \rightarrow P S
\]
Hall (and Goodenough and Gerhart) use these definitions to discuss notions of validity and reliability of testing methods. Our interest is in the model itself and its application to the model of interactive systems presented in the previous section.

In terms of our model of interactive systems above, we are interested in the relationship between a specification of type $M$ and a program which can also be modeled by type $M$.\(^3\) We define a type for test selection methods ($strategy_M$) and define an implementation relation $imp_M$ as a homomorphism from $s$ to $p$.

\[
strategy_M[E, S] \equiv M[E, S] \times M[E, S] \rightarrow F(F(S))
\]

\[
imp_M : [E, S] \leftrightarrow [M[E, S]]
\]

\[
\forall s, p : [M[E, S]]; e : E \mid e \in \text{dom } s.\text{actions} \bullet \exists f : S \leftrightarrow S \bullet
\]

\[
(s.\text{actions}(e) \triangleright f) = (f \triangleright p.\text{actions}(e))
\]

The implementation relation $imp_M$ defines exhaustive testing which is generally impossible to implement. We define a finite implementation relation $imp_M$ to permit practical testing. The predicate requires the existence of a representation relation $f$ and a finite set of input sets that correspond to test inputs. The set of test inputs is defined to be a partition with equivalence classes based on error detecting capability; the input space is partitioned in an attempt to partition the total computation. We define each element of each equivalence class of the input to produce the same result as the other members of the class. A representative of the class is distinguished as demonstrating conformance of the implementation to the specification.

\[
imp_M : [M[E, S]] \leftrightarrow [M[E, S]]
\]

\[
\forall s, p : [M[E, S]]; e : E \mid e \in \text{dom } s.\text{actions} \bullet
\]

\[
\exists f : S \leftrightarrow S; \{t\} \in \text{dom } (s.\text{actions}(e)) \bullet
\]

\[
\exists t : \{t\} \subset (s.\text{actions}(e) \triangleright f) = \{f \triangleright p.\text{actions}(e))\}
\]

The application of testing strategies is fundamental to the Test Template Framework which we describe in the next section.

\(^3\) Informally, the hypothesis of specification-based testing is that the program under test can be defined in the specification notation and that executing the program can reveal the conformance of the ‘program specification’ to the specification it implements.
3.2 Test Template Framework

The Test Template Framework (TTF) is a formal, abstract model of testing, used to derive a hierarchy of test information, including test inputs and outputs, from a formal specification [23]. The TTF provides a mechanism for recording a human tester’s decisions about testing, including strategies to be applied, coverage levels to achieve, etc. The TTF is not automated but provides a context for the use of testing tools such as test case generators, test execution frameworks and result comparators.

The root of the hierarchy for a particular operation is the inputs for which the operation is defined. The nodes of the hierarchy are constrained subsets of the input values of the operation. Test selection proceeds by selecting a test template (initially the entire valid input space) and applying a testing strategy to yield further test templates. The TTF does not prescribe the use of particular test selection techniques but, rather, advocates the use of multiple techniques by providing a means for structuring testing information.

The TTF uses the valid input space (VIS) of an operation as the source of all tests. The valid input space is that part of an operation’s input for which the operation is defined. In Z, the valid input space is the operation’s precondition, which is defined as the operation with after-state and output variables hidden. For an operation, Op, the valid input space is

\[ VIS_{Op} \subseteq \text{pre } Op \]

The basic unit for defining data in the TTF is a test template (TT) which is a constrained subset of the valid input space.

\[ TT_{Op} = \subseteq P VIS_{Op} \]

Testing information is derived using testing strategies. We take the set of all testing strategies as given.

\[ [\text{STRATEGY}] \]

Test templates for an operation are organised into a hierarchy called the test template hierarchy (TTH) which has the valid input space as root. The hierarchy is created by applying testing strategies to existing test templates to derive additional test templates. A test template hierarchy is declared for each operation under test.

\[ \text{TTH}_{Op} : TT_{Op} \times \text{STRATEGY} \rightarrow P \text{TT}_{Op} \]

In terms of the model of interactive systems, the TTF is used for testing single operations. The complete set of operations for a functionality or presentation view is represented by actions in schema M (with type \( E \rightarrow (S \leftrightarrow S) \)). For an operation \( o \), the valid input space is \( \text{dom}(\text{actions}(o)) \) and a test template is a subset of \( \text{dom}(\text{actions}(o)) \). Construction of the test template hierarchy for \( o \) consists of applying an instance of \( \text{strategy}_{M} \) to a test template to derive additional test templates. The derived test templates usually partition the states of the parent.
4 Behaviour-based SBT

In Sections 4.1 and 4.2 we discuss behaviour-based specification-based testing for implicit and explicit behaviour, and in Section 4.3 we extend the Test Template Framework to accommodate explicit behavioural notations. By implicit behaviour we mean constraints on operation sequencing imposed by a model-based specification \( M \), and, in contrast, by explicit behaviour we mean the use of a behaviour-based specification \( B \) to define an interactive view.

4.1 Implicit Behaviour

Dick and Faivre [2] present techniques for converting a model-based specification into a finite state machine for test sequencing and specification coverage. They form sub-operations by partitioning the state space by forming a canonical disjunctive normal form from operation predicates.

In terms of our model of interactive systems above, Dick and Faivre effectively derive a behavioural specification from actions in schema \( M \). Their technique constructs a finite state machine as a set of transitions \( F(FS \times EXFS) \) from a set of operations \( \langle as \rangle \) and a partition of the state space of the operations \( \langle tts \rangle \). For each transition, the label \( l \) corresponds to the event invoking the operation, and the before- and after states are both in \( tts \), and are in the input and output space of the operation, respectively.

\[
\begin{align*}
OPS[E,S] &= E \rightarrow (S \leftrightarrow S) \\
TT[S] &= F(FS) \\
TRN[E,S] &= F(FS \times EXFS)
\end{align*}
\]

\[
\begin{align*}
\forall as : OPS[E,S]; tts : TT[S] | \\
\cup tts &= \text{dom}(\bigcup \text{ran as}) \cup \text{dom}(\bigcup \text{ran as}) \land \text{tts} = \emptyset \\
\text{fsm}(as, tts) &= \{ \text{pre} : FS; l : E; \text{post} : FS | \\
& l \in \text{dom as} \land \\
& \text{pre} \in \text{tts} \land \\
& \text{post} \in \text{tts} \land \\
& \text{pre} \subseteq \text{dom}(\text{as}(l)) \land \\
& \text{post} \subseteq \text{ran}(\text{as}(l)) \}
\end{align*}
\]

The finite state machine defines possible sequences of operation invocations for functionality and presentation views. An interaction view, constructed as an explicit behavioural specification, further constrains system behaviour. In terms of our models, \( B \) traces imposes constraints beyond \( \text{fsm}(M\text{.actions}, \text{tts}) \) for some partition of \( M \)'s input space \( \text{tts} \).
4.2 Explicit Behaviour

There has been considerable work on using specifications written in behaviour-based notations to support conformance testing of communications protocols. The notations involved can be classified into (non-deterministic) finite state machines and a variety of notations with labelled transition systems (LTS) as the underlying semantic model. In this section we only model labelled transition systems as they are more general. This is the least developed aspect of this work.

We use $S \times L \times S$ as the type for a labelled transition. We define support functions to project the components of a transition.

<table>
<thead>
<tr>
<th>[S, L]</th>
</tr>
</thead>
<tbody>
<tr>
<td>before, after : $(S \times L \times S) \rightarrow S$</td>
</tr>
<tr>
<td>label : $(S \times L \times S) \rightarrow L$</td>
</tr>
<tr>
<td>$\forall b, a : S; l : L \ni$</td>
</tr>
<tr>
<td>$\quad \text{before}(b, l, a) = b \land \text{after}(b, l, a) = a \land \text{label}(b, l, a) = l$</td>
</tr>
</tbody>
</table>

A labelled transition system (LTS) consists of a finite set of states, a finite set of labels, a set of labelled transitions and a distinguished initial state.

<table>
<thead>
<tr>
<th>LTS[S, L]</th>
</tr>
</thead>
<tbody>
<tr>
<td>states : $\forall S$</td>
</tr>
<tr>
<td>labels : $\forall L$</td>
</tr>
<tr>
<td>trans : $\mathcal{P}(S \times L \times S)$</td>
</tr>
<tr>
<td>$s_0 : S$</td>
</tr>
<tr>
<td>$\text{states} = \text{before}[\text{trans}] \cup \text{after}[\text{trans}]$</td>
</tr>
<tr>
<td>$\text{labels} = \text{label}[\text{trans}]$</td>
</tr>
<tr>
<td>$s_0 \in \text{states}$</td>
</tr>
</tbody>
</table>

Many implementation relations have been explored for labelled transition systems including observational equivalence, bisimulation, testing equivalence, trace equivalence and failure equivalence. Our work in this area is at an early stage. As an example, we define a function traces to calculate the traces of an LTS. The function generates sequences of transitions that are within the LTS trans, with initial state $s_0$ and consecutive transitions state-related. The sequence of transitions is composed with label to yield a sequence of $L$ as required.

---

4 Our model does not distinguish $\tau$, the hidden or internal action.
\[\begin{align*}
[S, L] \\
\text{traces: } \text{LTS}[S, L] \rightarrow \text{P(seq } L)\\
\forall l : \text{LTS}[S, L] \cdot \text{traces}(l) = \{ts : \text{seq } (S \times L \times S)\} \\
\quad \text{ran } ts \subseteq l.\text{trans} \land \\
\quad \text{before}(ts(1)) = l.s_0 \land \\
\quad (\forall i : 2 \ldots \#ts \cdot \text{after}(ts(i - 1)) = \text{before}(ts(i))) \cdot \\
\quad ts \upharpoonright \text{label}
\end{align*}\]

We define \(\text{imp}_{\text{trace}}\) to model conformance of an implementation to a specification with respect to traces. Informally, an implementation \(i\) may only engage in traces in which its specification \(s\) could engage (which corresponds to observational compatibility for objects).

\[\begin{align*}
[S, L] \\
\text{imp}_{\text{trace}}: \text{LTS}[S, L] \leftrightarrow \text{LTS}[S, L]\\
\forall i, s : \text{LTS}[S, L] \cdot \\
\quad i \text{ imp}_{\text{trace}} s \iff \text{traces}(i) \subseteq \text{traces}(s)
\end{align*}\]

In addition to a variety of implementation relations, protocol testing also provides various testing approaches, many of which involve the use of a process (LTS) as tester. We have further work to do in this area. In the next section we introduce extensions to the Test Template Framework that incorporate trace-based tests.

### 4.3 Extended Test Template Framework

The original TTF records derived testing information for a particular operation. Interaction views specified in a behavioural notation require a framework that encompasses multiple operations structured, for example, as a sequence or trace of invocations. Two issues must be addressed in devising such a framework: signature compatibility and test case structuring.

*Signature compatibility* refers to assimilating the operations and their testing information to avoid type conflicts. We proceed to define a valid input space and test template type for each operation, for example \(Op_1\) and \(Op_2\). We then construct a generic operation by disjoining the individual operations.\(^5\)

\[
\begin{align*}
\text{VIS}_{Op_i} & \doteq \text{pre } Op_1 \\
\text{TT}_{Op_i} & \doteq \text{P VIS}_{Op_i}
\end{align*}
\]

\(^5\) Note that the signatures of the disjoined operations are merged to give the signature of the generic operation. An alternative approach which preserves the individual signatures is to use a free type as a disjoint union (at the cost of an additional level of indirection for schema references).
\[ \text{TEST} \]

\[
\begin{align*}
\text{TTH}_{Op_1} & : TT_{Op_1} \times \text{STRATEGY} \rightarrow P TT_{Op_1} \\
\text{TTH}_{Op_2} & : TT_{Op_2} \times \text{STRATEGY} \rightarrow P TT_{Op_2} \\
\text{TC} & : \text{STRATEGY} \rightarrow P(\text{seq } Op)
\end{align*}
\]

Test case structuring refers to the structuring of the derived testing information in test template hierarchies and as sequences of operation invocations. These sequences correspond to the testing information derived from a behavioural notation. We define a structure that combines the test template hierarchies of the operations with a set of test cases (TC), each of which is an operation sequence.

In terms of our model of interactive systems, we include a test template hierarchy for each operation in M. actions and test cases derived from B. traces are included in TC.

5 Conclusion

In this paper we presented an abstract model of specification-based testing of interactive systems. We began with a model of interactive systems that was based on models developed by Duke and Harrison [3, 4]. The model incorporates model-based and behaviour-based notations providing views of functionality, presentation and interaction. These models have much in common with Smith's models of objects [20–22].

We then modeled specification-based testing related to the model-based functionality and presentation views in the interactive system model. We examined previous testing theories based on a view of specifications and programs as functions [9, 7]. We modified this model to match our interactive system model and related it to the Test Template Framework.

Finally, we considered specification-based testing related to the behavioural notations in both implicit and explicit forms. For implicit behaviour we modeled Dick and Faivre’s technique for constructing a finite state machine from a model-based specification. We have used this technique for testing Object-Z classes using the ClassBench methodology [18]. Duke et al. [5] and Evans [6] use similar techniques to specify concurrent and reactive behaviour using Z. We also outlined models of testing based on explicit behavioural notations for protocol conformance testing: finite state machines and labelled transition systems. This is the least well-developed aspect of our work. We introduced extensions to the Test Template Framework to accommodate behavioural notations [17].

There is considerable scope for expanding our model of behaviour-based testing, particularly by covering more implementation relations which will have an
impact on the nature of the testing information generated. We also plan to investigate converting the models presented in this paper, which are based on event sequences, to partially ordered sets (posets) over events. This will permit investigation of non-interleaving concurrency. This may require incorporating the Test Template Framework modifications we proposed in [15], particularly deferring focusing on the input space for as long as possible.

We also plan to investigate the applicability of this model to Object-Z. Two versions of Object-Z offer multiparadigm capabilities that may be suitable for interactive system specification: the original variant with history predicates expressed in temporal logic, and Smith’s semantic integration of Object-Z and CSP [21, 22].

Formal specifications can be used to support testing of interactive systems. Models of such systems and such testing enable us to improve our framework for specification-based testing of interactive systems.

Acknowledgements

Ian MacColl is supported by an Australian Postgraduate Award and a Telstra Research Laboratories Postgraduate Fellowship.

References


A Superposition Refinement of Component-based Systems

Winnie Weiqun Qiu and John Zic
School of Computer Science and Engineering
University of New South Wales
NSW 2052

Abstract. We introduce a notion of superposition refinement as a method of constructing systems of reusable components. In this method, an initial specification is described in terms of a composition of the components and their interactions. A refinement of this specification is allowed to extend both the components and their interactions subject to the principle that the newly introduced interactions and extended components do not violate those established in the abstract specification.

1 Introduction

Component-based software systems require open and flexible conglomerations of distributed software components rather than monolithic heaps of code. System components are designed to be compositional, and computations can be flexibly composed from these interactive components [17]. This view of system construction has forced a paradigm shift away from algorithms and towards capturing component interactions and composing these various components while preserving desirable system characteristics such as correctness.

The preservation of desirable characteristics has also been approached by using stepwise refinement techniques. These techniques provide a method for the systematic construction of programs by applying a series of property preserving transformations (such as state invariants) from an initial abstract specification to a final set of concrete programs. Examples of well-known stepwise refinement techniques are the Refinement Calculus [9] and the B-Method [5]. The Refinement Calculus is based on the weakest pre-condition semantics [10], and has a typical specification expressed in terms of a system’s state space with pre-condition and post-condition relations on this state. The B-Method has the same theoretical foundation as the Refinement Calculus, although it adds some high-level constructs in order to guide the development of a final correct implementation from an initial specification.

These stepwise refinement approaches are not suited for reusable, component-based systems since they do not account for component interactions and compositions. First, refinement specifications typically describe component functionality yet hide its behaviours. If we refine one component into another, then this implies that we can reuse a component by knowing what it can do. However, we are
not allowed to access the state of the component directly, and so the refinement of a component is internal and independent of the context of the component. The second way that the stepwise refinement fails in component-based systems is that component interfaces may not be refined. If component \( C' \) is a refinement of component \( C \), \( C' \) only has the same operations as \( C \). Thus \( C' \) is better than \( C \) and it cannot do more than \( C \).

This paper introduces a method for deriving behavioural compositions in a refinement manner while preserving desirable properties. The proposed method supports both context-dependent refinement and interface refinement and extends the usual refinement notions with a hope in developing a better formal software technology for component-based systems. Our refinement technique is based on the concept of a superposition refinement, which supports the view that a software development process preserves the meaning of a specification while adding/imposing additional requirements [15]. For example, in a sequence \( S_0, \ldots, S_n \) of refinements, if \( S_{i+1} \) superposes \( S_i \) without violating the meaning of \( S_i \), then \( S_{i+1} \) is said to be a refinement of \( S_i \). \( S_0 \) would be the first approximate specification that satisfies basic but essential properties and \( S_n \) would be a final implementation. We see each specification as a consistent, behavioural composition of components and the refinement of components is carried out solely in the context of their composition. During a refinement process we allow interface extensions of the used components as well as possibly introducing new components into the composition.

In Section 2 we give a formal specification of behavioural composition based on action systems. The specifications are written in B-style Abstract Machine Notation [5]. The superposition is formalized by a set of refinement rules in Section 3 and a simple example illustrates its application in a bottom up development method. Section 4 compares our approach with other related works and sketches our prospects for future work.

## 2 Behavioural Specification

We specify a component-based system as an abstract dependency-oriented behavioural composition. The dependencies that determine how the components are put together correspond to the static structure of the system whereas its dynamic behaviour corresponds to the interactions between its components. The specification of a system is a description of certain invariant properties that define the conditions of the behavioural composition consistency.

A system is expressed in a notation based upon the B-Method’s Abstract Machine Notation and extended to allow the modelling of component interactions. This extension is based on Back’s action systems formalism [1]. Component interactions are embedded within an abstract machine specification. In this section, we also give an initial specification of a small case study, a presentation-semantics composition.
2.1 Notation

We specify behavioural composition in a machine notation which is similar the B-Method’s Abstract Machine Notation. It has a general form shown in Figure 1.

```
MACHINE
  the name of the machine
INCLUDES
  references to other machines being composed by this machine
SETS
  the names of deferred sets
DEFINITIONS
  the names of sets
VARIABLES
  Participant, Dependancy, Operation, Activity, Interface
INVARIANT
  INV
CONDITIONS
  CONS
INITIALIZATION
  which is used to establish invariants
INTERACTIONS
  defines a set of interations between participant components
END
```

Fig. 1. The B-Method’s Abstract Machine Structure

Except for the CONDITIONS clause and the INTERACTIONS clause, all other clauses have the same name and meaning as in the AMN of the B-Method.

The INCLUDES clause is an assembly primitive [6]. It composes the machines of components that participate the behavioural composition. Everything of a included machine is visible to the including machine, and the states of included machines are parts of the including machine’s state.

The VARIABLES clause introduces the state variables of the machine. The state is partitioned into five parts:
- The Participant variables represent the instances of the components,
- the Dependancy variable defines the connection of the components,
- the Operation variable is a set of operations responding to the messages from outside or inside of the machine,
- the Interface variable, through which the components have access to each other, is a subset of the operation set and
- the Activity variable represents the operations’ performance.

The INVARIANT clause consists of a number of predicates separated by the conjunction operator Λ. The predicates constrain the state variables.
The **INTERACTIONS** clause defines a set of interactions. An interaction is expressed in terms of operation invocations, and it specifies a synchronous event that requires a simultaneous participation of all the involved components. We will define the operation invocation and three operations for constructing interactions in Section 2.2.

The **CONDITIONS** clause introduces invariant properties of the behaviour of a machine. The conditions are specified by a group of predicates. All the interactions of a machine must satisfy the predicates in this clause to guarantee the behavioural composition consistency.

### 2.2 Specifying Interactions

An interaction \( \text{Inter}A_i \) is a cooperative action between two or more components and has the form

\[ \text{Inter}A_i \models G_i \rightarrow A_i \]

where \( G_i \) is a boolean expression over the machine state (the guard of \( \text{Inter}A_i \)) and \( A_i \) is an operation invocation expression (the body of \( \text{Inter}A_i \), see below). Whenever \( G_i \) is fired, \( A_i \) will occur.

We do not distinguish between the action of sending a message and the action of receiving a message. Thus invoking an operation, \( msg \), from component \( i \) to component \( j \) is described by an atomic action given as the triple \((i, j, msg)\). Moreover, operation invocation \((i, j, msg)\) triggers the state transition of component \( j \), that is component \( j \) enters a new state when it performs operation \( msg \).

We define three operations on these actions: selection \( + \), concatenation \( ; \), and parallel \( || \).

Let \((i, j, msg)R\) denote that a predicate \( R \) over some interaction variables has been preserved after the occurrence of \((i, j, msg)\). These three constructs then can have the following definitions:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>((i, j, m1) + (h, k, m2))</td>
<td>((i, j, m1)R \lor (h, k, m2)R)</td>
</tr>
<tr>
<td>((i, j, m1); (h, k, m2))</td>
<td>((i, j, m1); (h, k, m2)R)</td>
</tr>
</tbody>
</table>
| \((i, j, m1) || (h, k, m2)\) if \((i, j, m1) \cap (h, k, m2) = \emptyset\) then \((i, j, m1)R \land (h, k, m2)R\) otherwise \(((i, j, m1); (h, k, m2) + (h, k, m2); (i, j, m1))R\)

**Table 1.** Constructive Operations

Here \( i, j, h, k \) are the components involved in the interaction, and the messages \( m1, m2 \) are different.
The selection operator chooses one of the two alternatives, but no matter which one is selected, both lead to $R$ holding. The concatenation operator constructs an interaction from a pair of actions by simply putting them in sequential order. Once again, the result of this interaction is that $R$ holds. Finally, the parallel composition of two actions is either the simultaneous execution of the actions when none of the components is involved with both actions, or it is an interleaving of concatenation actions when any component can join more than one action.

An interaction specification results in a proof obligation about the preservation of the properties defined in INCOMPLETE and CONDITIONS. The Intera, presenting the INTERACTIONS clause, stands for a set of interactions

\[ \text{Intera} = \{ \text{Intera}_1, \ldots \text{Intera}_n \} \]

as well as for their logical disjunction

\[ \text{Intera} = \text{Intera}_1 \lor \ldots \lor \text{Intera}_n \]

Thus, if we can prove that

\[ \forall \text{Intera}_i \in \text{Intera} \bullet (\text{Intera}_i) \text{INV} \land ((\text{Intera}_i) \text{CONS} \lor \text{Intera}_i = \text{StutCONS}) \]

then the behaviour of a machine avoids internal divergence.

StutCONS denotes a "stuttering" interaction that is insensitive to the conditions specified by CONS. Notice that Intera is non-deterministic in allowing any of the enabled interactions to occur next, including stuttering interactions.

The model of Intera is close to the execution model of join action systems [2], where the body of an action is express as multiple assignment statements and the system properties are specified in temporal logic.

### 2.3 Example: a Presentation Semantics System

To illustrate our method, we specify a PresentationSemantics system. The system has a dependency among its components; that is, the component Presentation always reflects the change of the component Semantics. The behavioural composition based on this dependency is defined in Figure 2 and Figure 3.

The machine PresentationSemantics are composed by the machine Presentation and the machine Semantics and thus the state of the machine PresentationSemantics is a sum of states of both Presentation and Semantics. The presentation and semantics are Participant variables. Dependency variable, dependency, is a relation between the components. The p_activity and s_activity are Activity variable. The operations of the machine PresentationSemantics is an union of operations of Presentation and Semantics. Interface variable, like p_interface and s_interface, are global variables of the
MACHINE
Presentation
SETS
PRESENTATION;
P_OPERATION = {update, redraw};

P_DEPENDENT = {model}
VARIABLES
presentation,
p_dps, p_activity,
p_ops, p_interface
INVARIANT
presentation ∈ PRESENTATION\n p_dps ∈ presentation → P P_DEPENDENT\n p_ops ∈ presentation → P P_OPERATION\n p_activity ∈ p_ops(presentation)\n p_interface ⊆ p_ops(presentation)
INITIALIZATION
p_ops(presentation) := {update, redraw};
p_interface := {update};
p_dps(presentation) := {model}
INTERACTIONS
{ p_activity = update →
 (presentation, presentation, redraw) }
END

MACHINE
Semantics
SETS
SEMANTICS;
S_OPERATION = {getChange, setChange, notify};
S_DEPENDENT = {view}
VARIABLES
semantics,
s_dps, s_activity,
s_ops, s_interface
INVARIANT
semantics ∈ SEMANTICS\n s_dps ∈ semantics → P S_DEPENDENT\n s_ops ∈ semantics → P S_OPERATION\n s_activity ∈ s_ops(semantics)\n s_interface ⊆ s_ops(semantics)
INITIALIZATION
s_ops(semantics) := {getChange, setChange, notify};
s_interface := {getChange};
s_dps(semantics) := {view}
INTERACTIONS
{ s_activity = setChange →
 (semantics, semantics, notify) }
END

Fig. 2. Presentation and Semantics Machines
system through which the components are allowed to communicate, whereas the variable, interface, allow a "user" to activate the system.

The condition under the CONDITIONS clause,

\[ s_{\text{activity}} = \text{notify} \Rightarrow p_{\text{activity}} = \text{update} \]

defines a behaviour consistency. The machine PresentationSemantics will behave as the machine Presentation for the interactions when only invoking the operations of Presentation; similarly, it will behave as the machine Semantics when only invoking the operations of Semantics; but it will behave as the machine Presentation and the machine Semantic for the interactions when invoking both the operations of Presentation and Semantics. Thus the behaviour of PresentationSemantics is the composition of the behaviours of Presentation and Semantics with the condition \[ s_{\text{activity}} = \text{notify} \Rightarrow p_{\text{activity}} = \text{update} \].

MACHINE
PresentationSemantics
INcludes
Presentation, Semantics
Definitions
\( \text{PARTICIPANT} \triangleq \{ \text{presentation}, \text{semantics} \} \);
\( \text{OPERATION} \triangleq p_{\text{ops}}(\text{presentation}) \cup s_{\text{ops}}(\text{semantics}) \);
\( \text{DEPENDENT} \triangleq p_{\text{ops}}(\text{presentation}) \cup s_{\text{ops}}(\text{semantics}) \);
Variables
dependency, interface
Invariant
\( \text{interface} \subseteq p_{\text{interface}} \cup s_{\text{interface}} \land \)
\( \text{dependency} \in \text{PARTICIPANT} \Rightarrow (\text{DEPENDENT} \rightarrow \text{PARTICIPANT}) \land \)
\( \text{dependency}(\text{presentation})(\text{view}) = \text{presentation} \land \)
\( \text{dependency}(\text{semantics})(\text{model}) = \text{semantics} \);
Conditions
\( s_{\text{activity}} = \text{notify} \Rightarrow p_{\text{activity}} = \text{update} \);
Initialization
\( \text{dependency}(\text{semantics})(\text{view}) := \text{presentation}; \)
\( \text{dependency}(\text{presentation})(\text{model}) := \text{semantics}; \)
\( \text{interface} := \{ \text{getChange} \} \);
Interactions
\{ 
\( s_{\text{activity}} = \text{getChange} \rightarrow \)
\( \text{StartCONS}, \text{ InterA}_1 \)
\( s_{\text{activity}} = \text{notify} \rightarrow \)
\( (\text{semantics, presentation, update} \) \text{ InterA}_2 \)
\}
End

Fig. 3. The PresentationSemantics machine
Notice that the specifications might only specify essential behavioural requirements and underspecify some inessential ones. As the development proceeds, without destroying already included behaviours, the specifications can be superposed with more detail. This is an incremental specification method.

3 Refinement

3.1 Superposition Refinement

The term, superposition, was introduced by Dijkstra in [11]. Superposition is a design method where an initial solution is reasonably small and satisfies some basic requirements—even a single design decision, and then a sequence of successive extensions is performed without destroying the properties that have already been certain. Recently superposition has been proposed as an approach to the refinement of action systems [1]

The superposition defined here allows to extend the state of a machine both by extending operations of the machine and by extending dependencies of the machine. What is essential is that the behaviours of a refined machine always include the behaviours of the original machine.

Consider the two machine $C$ and $C'$

**Definition 1.** $C'$ is a superposition refinement of $C$ if and only if

1. $C'$ has new variables $P'$, $D'$, $O'$, $A'$ and $I'$ in addition to the variables $P$, $D$, $O$, $A$ and $I$ that $C$ has. $P$ and $P'$ denote all participant variables. $D$ and $D'$ denote all dependency variables. $O$ and $O'$ denote all operation variables, $A$ and $A'$ denote all activity variables and $I$ and $I'$ denote all interface variables,

2. each interaction inter$A'_i$ is an extension of inter$A_i$, with

   (a) $A'_i$ holds INV $\wedge$ INV'
   (b) $A'_i$ holds CON $\wedge$ CON'
   (c) $G'_i$ strengthens $G_i$
   (d) $\forall bhv \in BHV(A_i), \exists bhv' \in BHV(A'_i) \bullet$
   $bhv' \uparrow \{act \mid act \in operands(A_i)\} = bhv$

3. each interaction inter$B_j$ is a new interaction, with

   (a) $B_j$ holds INV'
   (b) $B_j$ holds CON'
   (c) $\forall bhv \in BHV(B_j) \bullet bhv = StuINV \wedge StuCON$

Intuitively, one can see that $C'$ has more detail than $C$ and these details are non-corrupting.

**Proposition 1.** If $C'$ is a superposition refinement of $C$, all safety properties of $C$ are satisfied in $C'$

$1 \uparrow$ is a sequence operator. If $t$ is a sequence and $A$ is a set, $(t \uparrow A)$ denotes $t$ is restricted to $A$ and results in a new sequence simply by omitting all elements outside $A$. 

MACHINE
C
VARIABLES
P, D, O, A, I
INVARIANT
INV
CONDITIONS
CON
INTERACTIONS
\{
G_1 \rightarrow A_1,
\ :
G_i \rightarrow A_i,
\ :
G_m \rightarrow A_m
\} 
\{
G'_1 \rightarrow A'_1,
\ :
G'_i \rightarrow A'_i,
\ :
G'_m \rightarrow A'_m
\} 
\{
Gk'_1 \rightarrow B'_1,
\ :
Gk'_i \rightarrow B'_i,
\ :
Gk'_m \rightarrow B'_m
\}
END
Madame.

Fig. 4. Superposition Refinement
There are different kinds of superposition design: a superposition mechanism should preserve all safety and liveness properties [7], only the preservation of safety properties is guaranteed but liveness properties need separate proof [8], and both properties are not automatically preserved [12].

In our specifications safety properties [14] are the most important ones and we have recommended relaxing transformation rules, so we support the second alternative. The third option does not seem to be a proper refinement.

As an example, consider how the machine PresentationSemantics can deal with failed updating. From its operation set, we can see the machine has no strategy to handle the rejection. Starting with extending OPERATION by the set \{request, accept, nonAvailable, reportRefusal\}, we have a superposed machine in Figure 5.

\begin{verbatim}
MACHINE PresentationSemantics'
INCLUDES ...
SETS
E_ops = \{request, accept, nonAvailable, reportRefusal\}
DEFINITIONS ...
VARIABLES ...
\textit{extend\_activity}
INVARIANT ...
\textit{extend\_activity} \in E_ops
CONDITIONS ...
INITIALIZATIONS ...
INTERACTIONS 
\{
\textit{extend\_activity} = accept \land s\_activity = notify \rightarrow
\textit{semantics}, presentation, update\}, \ (InterA')
\textit{s\_activity} = getChange \rightarrow
\textit{semantics}, presentation, request\}; \ (presentation, semantics, accept)\rightarrow
\textit{presentation}, semantic\_set, nonAvailable\}, \ (InterA)'
\textit{extend\_activity} = nonAvailable \rightarrow
\textit{semantics}, semantics, reportRefusal\} \ (InterB)
\}
END

Fig. 5. A superposition of machine PresentationSemantics
\end{verbatim}

The ellipses in Figure 5 represent the parts taken from the machine PresentationSemantics.
The new interaction \( \text{Inter}B_1 \) will not produce any behaviour sequences that cause modification of the variables of the machine \( \text{PresentationSemantics} \). The \( \text{Inter}A'_1 \) is an extension of \( \text{Inter}A_1 \) with a stronger guard. The body of \( \text{Inter}A'_2 \) is a refinement of \( \text{StmtCONS} \) in the body of \( \text{Inter}A_2 \) where the failed situation was not concerned.

We could have the machine \( \text{PresentationSemantics}' \) is a superposition refinement of the machine \( \text{PresentationSemantics} \) if all new operation invocations can be processed by the included machine \( \text{Presentation} \) and \( \text{Semantics} \).

### 3.2 Context-dependent Component Refinements

The monotonicity of the refinement allows us to replace a component \( A \) in a composition \( C(A) \) with another component \( B \) if \( B \) is a refinement of \( A \). However, the refinement of a composition could require all of the components to be refined at the same time, and not only justified by monotonicity alone.

As we have seen the variable of composition \( C(C_1, \ldots, C_k, \ldots, C_n) \) is composed by the variables of \( C_1, \ldots, C_k, \ldots, C_n \). Given a superposition on \( C \), partitioning the extended variable of \( C \) is assumed. For instance, we can partition \( E_{\text{ops}} \) of \( \text{PresentationSemantics}' \) into \( \text{Presentation} \) and \( \text{Semantics} \). The machine \( \text{Presentation}' \) and \( \text{Semantics}' \) are superpositions of the original ones. Notice that superposition on \( \text{PresentationSemantics}' \) is the superposition on the interface of \( \text{Presentation} \) and \( \text{Semantics} \).

In the context of \( C(C_1, \ldots, C_k, \ldots, C_n) \), let \( I_{C_k} \) be the interface of \( C_k \) and \( E \) be the extended variables of \( C \), we have a definition of the interface refinement.

**Definition 2.** \( C'_k \) is a superposition interface refinement of \( C_k \), if

1. there is \( C'(\ldots, C'_k, \ldots) \) that is a superposition of \( C(\ldots, C_k, \ldots) \)
2. \( I_{C'_k} \setminus I_{C_k} \subseteq E \)
3. in \( C' \) and \( C'_k \), for all the interaction, \( \text{Inter}A'_i \) that is related to \( op \in I_{C'_k} \setminus I_{C_k} \), is one of the follows:
   (a) an extension of \( \text{Inter}A_i \) in \( C \) or \( C_k \)
   (b) a new interaction \( \text{Inter}B_j \) in \( C' \) or \( C'_k \)

Obviously machines \( \text{Presentation}' \) and \( \text{Semantics}' \) are the superposition interface refinements of machines \( \text{Presentation} \) and \( \text{Semantics} \). We can rewrite machine \( \text{PresentationSemantics}' \) in Figure 7, which now is a superposition refinement of machine \( \text{PresentationSemantics} \).

Generalizing the above case, we obtain

**Definition 3.** \( C'(C'_1, \ldots, C'_n) \) is a superposition refinement of \( C(C_1, \ldots, C_n) \) if

1. \( C' \) is a superposition of \( C \)
2. \( C'_1, \ldots, C'_k \) are superposition interface refinements of \( C_1, \ldots, C_k \).
Fig. 6. Superposition of the two machines Presentation and Semantics
MACHINE
PresentationSemantics's
INCLUDES
Presentation', Semantic'
DEFINITIONS

VARIABLES

INVARIANT

CONDITIONS

INITIALIZATION

INTERACTIONS
{
  s_activity = notify \rightarrow
  (semantics, presentation, update),
  s_activity = getChange \rightarrow
  (semantics, presentation, request);
  (presentation, semantics, accept) + (presentation, semantic, nonAvailable)
}
END

Fig. 7. A superposition refinement of machine PresentationSemantics
4 Discussion

In many works, action systems and stepwise refinement are combined to provide a method for program construction. Back and Sere [3] used weakest preconditions refinement calculus for changing an initially large-grained sequential action system into a fine-grained and highly concurrent one by the refinement of action atomicity. In related work, Back [1] show how the superposition refinement rules for action systems can be formalized within the Refinement Calculus. In Kurki-Suonio’s work [13], temporal logic and action systems have been used as the fundamentals of object-oriented specification of collective behaviours and superposition refinement have been applied to reasoning about class multiple inheritance.

Most researches on composition of action systems dealt exclusively with reactive systems and relied upon the monotonicity of refinement. Both Back [1] and Kurki-Suonio [4] specified the implementation environment as an action system and compose it with the action system to be implemented in context of that assumed environment. The partitions of variables and actions are disjoint subsets. The refinement of the closed system or modular system is independent refinements of the composed action systems.

With the combination of action systems and superposition refinement, we began trying to refine behavioural compositions in order to formally support the construction of component-based systems. Our construction strategies are based on the notion that a system is a behaviour composition of its components. We use action systems as a basis for specifying behaviour composition. A system construction starts with behaviour composition at program level—existing components are combined to get a behaviour composition. Superposition has been used to extend the behaviour composition at specification level. We recommend relaxing the principle of monotonicity in order to allow non-monotonic refinement with the preservation of behavioural composition consistency. The refinement of individual components is not conducted independently but in context of their composition.

Our initial work appears to be satisfactory. More works need to be done to assess our ideas. In particular, we intend to formally extend the B Abstract Machine Notation to facilitate behavioural specifications, and to use B-Toolkit to support proofs of the preservation of consistency. Further, and perhaps more difficult, work involves the generation of proof obligations of liveness properties. Context-dependent refinements and interfaces refinements need a lot of further study as well.

Two interesting works discussed the concept conservative extension. Turski [15] interpreted the conservative extension as a formal basis for program refinement: a specification is more than its implementation and a software development process is preserving the meanings of the specification and adding execution-oriented details. Antoniou [16] investigated the interplay between the conservative extension property and non-monotonicity. We have seen the close correspondence between superposition and conservative extension. Both of these
works strengthen our belief that the approach that we are investigating will lead
to a useful technique for the construction of component based systems.

References

1. R. J. R. Back and K. Sere. Superposition refinement of reactive systems. Formal
2. R. Kurki-Suonio. Operational specification with joint actions: serializable
4. R. Kurki-Suonio. Refinement and composition in TLA-based closed-system speci-
fication. Technical Report, Software Systems Laboratory Tampere University of
   Press, 1996.
   In Proceedings of the Fifth Australian Refinement Workshop, Brisbane, Australia,
   April 1996.
8. L. Bourque and N. Francez. A Compositional approach to superposition. In Pro-
   ceedings of the 14th ACM Conference on Principles of Programming Languages,
   California, USA, January 1988.
9. C. C. Morgan and K. A. Robinson. Specification statements and refinement. IBM
    On-the-fly garbage collection: an exercise in cooperation. Communications of the
12. S. Katz. A Superimposition control construct. ACM Transactions on Programming
13. R. Kurki-Suonio. Fundamentals of Object-Oriented specification and modeling of
    collective behaviours Technical Report, Tampere University of Technology, Tampere,
    Finland 1997.
14. B. Alpern and F. Schneider Recognizing safety and liveness. Distributed Comput-
15. W. Turski and T. Maibaum The Specification of Computer Programs Addison-
16. G. Antoniou, C. MacNish and N. Foo Conservative expansion concepts for default
    theories. In Proceedings of 4th Pacific Rim International Conference on Artificial
    Intelligence, Springer, 1996.
17. O. Nierstrasz and D. Tsichritzis Object-Oriented Software composition Prentice
    Hall, 1995.
I Went Down To The Crossroads:
Conjoining Catamorphisms

Martin Schwenke

Department of Computer Science
The Australian National University
Martin.Schwenke@cs.anu.edu.au

Abstract. This paper attempts to address software specification, design
and implementation reuse by bringing together work from a number of
areas of program development. Refinement calculi are based on wide-
spectrum languages that include abstract, logical specifications, which
are transformed into executable programs. Functional and relational cal-
culi allow programs and specifications to be manipulated using higher-
order operations in powerful algebraic setting. Promotion is used in the
Z specification notation to allow simple operations to be reused within a
more complex framework. Specification conjunction has also been used
in refinement calculi as an aid to reuse. We provide a series of examples
that bring various elements of these areas together. Our examples cen-
tre on programs that can be elegantly specified by a calculation of all
permutations of an input list, combined with some restriction on these
permutations. This paper represents work in progress, so some of the ex-
amples are incomplete and, therefore, do not serve as convincing positive
examples of the method. Also, much of the work has not been completely
formalised. Our main contribution is to show that the combined approach
can work in some cases and, when it does work, it is extremely profitable.

1 Introduction

imperative programs to be developed that are correct with respect to their spec-
cifications. The specifications are usually logical specifications, containing predi-
cates that constrain their behaviour. Although these calculi meet their goal of
making program development a mathematical activity, they operate at quite a
low level of abstraction, and we believe that they are difficult to use.

Functional programming languages allow programs to be written at a higher
level of abstraction that imperative languages. According to Hughes [8], one rea-
son for this is the use of higher-order functions. Functional programming calculi,
such as Bird [4], are based on higher-order functions and contain correspond-
ingly high-order rules for manipulating programs. One reason for this is that
higher-order functions provide a level of indirection when they operate on data
structures, allowing for 'pointless' rules that deal with compositions of functions
rather than the application of functions to variables. Relational calculi, such as
that of Aarts, Backhouse, et al. [1], add nondeterminism and also provide a very high-level of abstraction. We believe that the elegance and level of abstraction found in these calculi are due to a profound understanding of the data structures involved.

There has been a substantial amount of work on refinement calculi targeted at functional programming languages (expression refinement calculi; for examples, see Norvell and Hehner [14], Ward [21], Schwenke and Robinson [17], Bunkenburg [5], Morris [13], Mahony [10], Schwenke and Mahony [16]). However, we believe that none of this work attempts to exploit the use of higher-order functions, based on known data structures, to help make refinement easier.

In the Z notation (Spivey [18]) low-level operations that operate on some type are often promoted so that they operate on collections involving that type. This method of specification reuse is similar to the way that functional languages allow operations to be ‘plugged’ into the framework provided by higher-order functions. Another method of combining specifications, which is also found in Z, is conjunction. Ward [20] and Groves [6] have explored the use of conjunction in the refinement calculus as a means of building specifications from components, and possibly reusing components. Mahony [9] provides a theoretical basis for conjunction in the refinement calculus and explores the relationship between conjunction and promotion.

We provide a series of examples involving all of these ideas. The style of presentation if fairly informal. We assume that we have an expression refinement calculus that provides all of the standard, useful refinement rules, like weakening preconditions and strengthening postconditions. Most of the expression refinement calculi mentioned above are at least partially suitable. We introduce notation and describe semantics informally. We hope to convince the reader that our examples would not be much more difficult if they were presented formally. Some of the examples are incomplete, and it is unclear whether they could actually be completed in the desired style.

2 Conjunction

Conjunction operators are well known in specification notations, but are less familiar in programming languages. We use $\cap$ to represent conjunction across our wide-spectrum language. The meaning is obvious if relational (or set-based) semantics are used, and is fairly intuitive for predicate transformer semantics. In particular, conjoining two function specifications means conjoining their preconditions and postconditions respectively.

\[
(\lambda x \bullet P_1 \Rightarrow \cap y[R_1]) \\
\cap \\
(\lambda x \bullet P_2 \Rightarrow \cap y[R_2]) = (\lambda x \bullet \left( \begin{array}{c}
P_1 \\
P_2 \\
\end{array} \right) \Rightarrow \cap y \left( \begin{array}{c}
R_1 \\
R_2 \\
\end{array} \right))
\] (1)

In the above function specifications, $P_i$ represents a precondition that restricts input $x$, and the output $y$ is chosen to satisfy a postcondition $R_i$. 
We also require conjunction to be monotonic with respect to refinement.

\[ a \sqsubseteq a', \ b \sqsubseteq b' \]

\[ a \sqcap b \sqsubseteq a' \sqcap b' \] \hspace{1cm} (2)

Our belief is that conjunction can be put to good use, ‘welding’ together programs of identical structure. This is particularly true when programs are written using higher-order operations. More generally, if two programs can be written using a common recursive or iterative framework, then the conjunction of the two can be written using a single instance of that framework. That is, function application distributes weakly over refinement.

\[ f \ a \sqcap f \ b \sqsubseteq f \ (a \sqcap b) \] \hspace{1cm} (3)

We can take \( f \) to be a higher-order function that represents the common framework. The above condition holds under relational semantics even if \( f \) is a relation instead of a function. It probably also holds under predicate transformer semantics.

3 A Singular Insertion Sort

We provide a specification of sorting, and provide a high-level derivation that produces an insertion sort algorithm. We characterise the derivation in this section as being singular because we specify a program that produces a single output and describe how it is related to its input. This contrasts with the derivation in the next section, which explicitly uses set-valued functions.

We begin with a specification of \( \text{sort} \), as follows:

\[ \text{sort} :: (\text{Ord } \alpha) \Rightarrow [\alpha] \rightarrow [\alpha] \]

\[ \text{sort} = \text{perm} \cap \text{ascendingOutput} \] \hspace{1cm} (4)

The notation used is based on Haskell [15]. The function \( \text{sort} \) is defined between lists of elements \([\alpha]\). The elements can be of any type that has an ordering \((\text{Ord } \alpha)\), permitting operations such as < (less than). We use the algebraic ‘cons’ lists found in functional programming languages, rather than the function-based sequences found in \(Z\). An informal interpretation of the above specification is that to sort a list we must produce an ascending permutation of the input.

3.1 Defining and Refining Permutations

The relation \( \text{perm} \) takes a list and returns a list containing the same items as the original.

\[ \text{perm} :: [\alpha] \rightarrow [\alpha] \]

\[ \text{perm} = (\lambda \text{xs} \bullet (\cap \text{ys} \mid \text{items} \ \text{ys} = \text{items} \ \text{xs})) \] \hspace{1cm} (5)
The body of the λ-abstraction is a generalised choice, or a logical specification of an expression. Note that although we are using ‘cons’ lists we use a Z-style items function to convert lists to bags.

We assume that we can implement perm using foldr. foldr is probably the best known example of a catamorphism, which is a special type of operation that employs recursion over an inductive datatype.

\[
\text{perm} \subseteq \text{foldr include} \ [\]
\]

The initial value of the computation is the empty list, [], and foldr moves along the input list, using the accumulation relation, include, to add each item to its output.

The specification of include is quite relaxed. It insists that all the items in the input list are present in the output, along with the new item being included.

\[
\text{include} :: \alpha \to [\alpha] \to [\alpha] \\
\text{include} = (\lambda x \ x s \bullet (\exists \ y s \ | \ \text{items} \ y s = [x] \uplus \text{items} \ x s))
\]

If we are calculating permutations, it seems that an efficient refinement of include might be one that doesn’t do any permuting of the partial results! That is, we can replace include by a specification that maintains the order of items in the input list, but can put the new item in any position. We call this relation inject.

\[
\text{inject} :: \alpha \to [\alpha] \to [\alpha] \\
\text{inject} = (\lambda x \ x s \bullet (\exists x s_1, x s_2 \bullet x s = x s_1 ++ x s_2))
\]

We claim that

\[
\text{include} \subseteq \text{inject}
\]

and need to prove that the postcondition has been strengthened, but leave this as an exercise.

The simplest and most efficient refinement of inject is (:) (cons), which always chooses z_1 to be empty, thus adding the new value at the beginning of the input list. This results in a very boring permutation, since:

\[
\text{foldr} (:) [] = \text{id}
\]

That is, implementing perm by refining include to (:) simply produces the identity function. All that we have found is that a list is a permutation of itself. Although we have taken the refinement of perm past the point where it can be part of an implementation of sort, we have some useful intermediate results. Also, even though the final step is not terribly interesting, it is a good sanity check.
3.2 Ascending Output

The relation \textit{ascendingOutput} ignores its input, but always outputs a list that is in ascending order.

\[
\text{ascendingOutput} :: (\text{Ord } \alpha) \Rightarrow \beta \rightarrow [\alpha] \\
\text{ascendingOutput} = (\lambda z \bullet (\exists ys \mid \text{isAscending } ys))
\]

(11)

For completeness we define \textit{isAscending} and provide an implementation.

\[
is\text{Ascending} :: (\text{Ord } \alpha) \Rightarrow [\alpha] \rightarrow \mathbb{B}
\]

\[
is\text{Ascending} = (\lambda xs \bullet (\forall i, j \mid 0 \leq i < \text{length } xs - 1 \bullet xs!!i \leq xs!!(i + 1)))
\]

\[
is\text{Ascending} \subseteq (\lambda xs \bullet \text{all (zipWith } (\leq) xs \text{ (tail } xs)))
\]

(12)

3.3 A Refinement Rule for \textit{foldr}

The specification of \textit{sort} (4) suggests that we wish to strengthen an implementation of \textit{perm} so that its output satisfies \textit{isAscending}. This leads us to investigate how we can refine a given specification by a \textit{foldr} operation. Our investigations lead us very quickly to an instantiation of the structural induction proof rule for lists.

\[
P(i), \quad (\forall x, xs \bullet P(xs) \Rightarrow P(f x xs))
\]

\[
(\forall ys \bullet P(\text{foldr } f i ys)) \quad \frac{P(i), \quad (\forall x, xs \bullet P(xs) \Rightarrow P(f x xs))}{(\forall ys \bullet P(\text{foldr } f i ys))}
\]

(13)

We can rewrite this as a refinement rule by instantiating the variables in the consequence with specifications that satisfy the conditions in the assumption.

\[
(\lambda xs \bullet (\exists ys \mid P(ys))) \quad \subseteq \quad \text{foldr } (\lambda x xs \bullet P(xs) \Rightarrow \exists ys \mid P(ys)) \quad (\exists i \mid P(i))
\]

(14)

That is, we begin with a specification of a function, mapping a list to a list, where the result satisfies \textit{P}. We can refine this by a \textit{foldr} operation, as long as the initial value satisfies \textit{P}, and the accumulation function maintains \textit{P}.

3.4 An Ascending Permutation

We can now implement \textit{ascendingOutput} (11) via (14).

\[
\text{ascendingOutput} \subseteq \text{foldr } (\lambda x xs \bullet \text{isAscending } xs \Rightarrow \exists ys \mid \text{isAscending } ys) \quad (\exists i \mid \text{isAscending } i)
\]

(15)

It is easy to prove that the initial value is refined by the empty list.

\[
(\exists i \mid \text{isAscending } i) \quad \subseteq \quad []
\]

(16)
We have now refined both conjuncts of (4). We choose \textit{inject} as a suitable refinement of \textit{perm}.

\[
\text{sort} \sqsubseteq \text{foldr} \left( \lambda \ x \ \text{xs} \ . \ \left( \sqcap \ \text{ys} \ . \ \left( \exists \ \text{x1}, \ \text{x2} \ . \ \text{xs} = \text{x1} + + \text{x2} \ \land \ \text{ys} = \text{x1} ++ \ [\ x ] ++ \ \text{x2} \right) \right) \right) \ \sqcap \\
\text{foldr} \left( \lambda \ x \ \text{xs} \ . \ \text{isAscending} \ \text{xs} \ \Rightarrow \ \sqcap \ \text{ys} \ . \ \text{isAscending} \ \text{ys} \right)
\]

We can use (3) to produce a refinement involving a single use of \textit{foldr}

\[
\text{sort} \sqsubseteq \text{foldr} \ \text{insert}
\]

and define \textit{insert} as the conjunction of the two accumulation relations, as follows.

\[
\text{insert} = \lambda \ x \ \text{xs} \ . \ \text{isAscending} \ \text{xs} \ \Rightarrow \\
\left( \sqcap \ \text{ys} \ . \ \left( \exists \ \text{x1}, \ \text{x2} \ . \ \text{isAscending} \ \text{ys} \land \text{xs} = \text{x1} + + \text{x2} \ \land \ \text{ys} = \text{x1} ++ \ [\ x ] ++ \ \text{x2} \right) \right)
\]

A reasonable implementation of \textit{insert} is

\[
\text{insert} \sqsubseteq \left( \lambda \ x \ \text{xs} \ . \ \left( \sqcap \ \text{ys} \ . \ \left( \exists \ \text{x1}, \ \text{x2} \ . \ \left( \text{xs} (\text{x1}, \ \text{x2}) = \text{break} (> \ x) \ \text{xs} \right) \ \land \ \text{ys} = \text{x1} ++ \ [\ x ] ++ \ \text{x2} \right) \right) \right)
\]

where \textit{break} splits its input list before the first element that satisfies the given condition.

4 A Plural Insertion Sort

We use a \textit{plural} version of insertion sort to motivate further discussion. By plural we mean that the set of all possible results is returned, which doesn’t make much of a difference for sorted lists, since all results will be the same. However, there are many problems where it is useful to consider a set of possible solutions rather than a single solution. For example, we might be interested in calculating all possible permutations of a list instead of a just a single permutation.

Fortunately, we have at our disposal \textit{A}, the \textit{power transpose} operator (Bird and de Moor [3]), which takes a relation and turns it into a set-valued function. For the example of permutations we can simply write

\[
\text{perms} = \text{A} \ \text{perm}
\]

to describe the function that returns the set of all permutation of a given list.

Even more fortunately, we have a simple rule for calculating the power transpose of a relation that is expressed in terms of \textit{foldr}.

\[
\Lambda (\text{foldr} \ f \ \text{i}) = \text{foldr} \left( \lambda \ x \ . \ \sqcap \ \circ \ \text{mapSet} (\Lambda (f \ x)) \right) \{ \text{i} \}
\]
This is an instance of a more general rule for relational catamorphisms given by Bird and de Moor [3].

If we consider that
\[(\lambda x)\{\text{insert } x \text{ xs} \} = (\text{singleton } \circ \text{insert } x \text{ xs})\]
then \(A\text{sort}\) can be written as follows.
\[A\text{sort} \subseteq \text{foldr} \left( \lambda x \cdot \cap \circ \text{mapSet} (\text{singleton } \circ \text{insert } x) \right) \{\} \]

We have shown that we can phrase an initial specification in the singular and still construct a plural implementation.

5 Filtering Permutations

We can generalise from the specification of \(\text{sort}\) given in (4) to a more generic shape of specification. Consider specifying a function \(f\) via the conjunction of a function \(g\) that produces a list, and an extra requirement on the output list \(P\).

We can write this as
\[f = g \cap (\lambda x \cdot (\cap y :: P(y)))\]

We can implement \(A\!f\) as follows.
\[A\!f \subseteq \text{filterSet } P \circ A\!g\]

That is, we can calculate all of the possible outputs of \(g\), and then discard those outputs that don’t satisfy \(P\). Groves [6] has proved some similar, more general results to do with implementing conjunctions using sequential composition in the imperative refinement calculus.

We could have used this as a very naive, though very simple, approach for implementing \(\text{sort}\).
\[A\text{sort} \subseteq \text{filterSet isAscending } \circ \text{perms}\]

The best case complexity is quite good when combined with laziness!

Focusing on the body of the accumulation relation in (24) we can observe the following property.
\[A\text{insert } x \subseteq \text{filterSet isAscending } \circ A\text{inject } x\]

By generalising \(\text{isAscending}\) to an arbitrary predicate \(P\), we get
\[\begin{align*}
\text{perms } \cap (\lambda x \cdot (\cap y :: P(y))) & \subseteq \\
\text{foldr} \left( \lambda x \cdot \cap \circ \text{mapSet} (\text{filterSet } P \circ A\text{inject } x) \right) \{\}\end{align*}\]

This is quite a useful general result, and is much more efficient than calculating all permutations and then filtering them.
6 Eight Queens

The eight queens problem is well known to most computer scientists. The goal is to place eight queens on a chess board so that no queen can attack another. Since a queen can move either horizontally or vertically, like a rook, or diagonally, like a bishop, we can use conjunction to specify the problem.

\[
\text{eightQueens} = \text{eightRooks} \cap \text{eightBishops} \tag{30}
\]

The eight rooks problem is easy to specify.

\[
\text{eightRooks} = \text{zip} (\text{perm}(['a..' 'h'])) (\text{perm}[1..8]) \tag{31}
\]

We take the eight distinct files and permute them, do the same for the ranks, and then construct pairs.

Since \(\text{perm} \subseteq \text{id} \) we can decide not to permute the files.

\[
\text{eightRooks} \subseteq \text{zip} (['a..' 'h']) (\text{perm}[1..8]) \tag{32}
\]

If we ignore the order of the pairs in the resulting list, this still leaves use with full coverage of the solution space. Refining out the \(\text{perm}\) of the ranks would leave us with a trivial solution where the rooks are arranged along the diagonal from (‘a’, 1) to (‘h’, 8).

The eight bishops problem doesn’t appear to have as simple a solution as the eight rooks problem. A solution can be specified using a list of pairs of ranks and files, of length eight, where no two pairs occupy a common diagonal. We won’t formalise this here, because we aren’t confident that there is a useful solution involving such a formalisation. Another problem is that we haven’t expressed the eight rooks problem in a form where \texttt{foldr} appears at the outermost level.

In conclusion, even though it is quite easy to express the eight queens problem using conjunction, there isn’t necessarily a refinement sequence that maintains the conjunction for any length of time. However, it is still possible that there is such a solution.

7 Boggle

7.1 Introduction

Boggle is Parker Brothers trademark for its hidden word game. The object of the game is to find words in a four-by-four grid. The positions in the grid are occupied by dice that have a letter of the alphabet on each of their six faces. Before each round the dice are shaken into the grid in a random arrangement. Words are formed by moving vertically, horizontally or diagonally between the topmost faces of the dice. Each die may only be used once in each word. The problem that we are interested in is finding all possible words in a particular grid, relative to some dictionary.
7.2 Grids and Paths

Before considering the letters associated with the dice, it is useful to consider the paths that can be constructed between them. We begin by constructing a four-by-four grid.

\[
\text{grid} \ = \ \{(x, y) \mid x \in [1.4], y \in [1.4]\}
\]  

(33)

A very naive way of specifying a solution is to consider all possible permutations of the grid elements, and restrict ourselves to those that are properly connected. We can then calculate all possible subsequences of each of these complete paths. In addition, we choose to work in a singular setting, specifying a single path.

\[
\text{path} \ = \ (\text{subsequence} \circ (\text{perm} \cap \text{outputConnected})) \text{grid}
\]  

(34)

We consider the specification of subsequence to be uninteresting, so we turn our attention to specifying connected paths through the grid. We could do this via a simple Boolean test but, for reasons that will become obvious, we choose to do this by calculating the length of a path by considering the distance between each pair of path elements.

\[
\text{pathLength} p \ = \ \sum i \mid 0 \leq i < \text{length} p - 1 \bullet \\
\max \ \text{abs} \ ((\text{fst} (p!!i) - \text{fst} (p!!(i + 1)))) \\
\min \ \text{abs} \ ((\text{snd} (p!!i) - \text{snd} (p!!(i + 1))))
\]  

(35)

Now we can specify a connected path as one whose pathLength is less than the number of elements in the path.

\[
\text{isConnected} p \ = \ \text{pathLength} p < \text{length} p
\]  

(36)

\[
\text{outputConnected} \ = \ (\lambda \ zs \bullet (\forall ys \mid \text{isConnected} ys))
\]  

(37)

In fact, the pathLength will be exactly one less that the length, but the above, more general definition turns out to be more useful. Also, note that isConnected handles any list of pairs of numbers, and isn’t restricted to work with grid.

If we choose to implement outputConnected using foldr, via (14), we encounter a fairly serious problem. Although we require the final result to be connected, we don’t necessarily want each intermediate result of foldr to be connected. That is, an intermediate result may not be connected, but may become connected if new elements are ‘injected’ into useful places. This is akin to choosing a loop invariant that is too strong, so possibly useful intermediate results are likely to be rejected.

We can provide a weaker version of isConnected that does the same job.

\[
\text{maxLength} \ = \ 16
\]  

(38)

\[
\text{isPossiblyConnected} p \ = \ \text{pathLength} p < \text{maxLength}
\]  

(39)

This condition only causes a partial path to be rejected if its length already exceeds the maximum allowable length. For complete paths (of length 16),
this is equivalent to \textit{isConnected}. However, this condition rejects fewer of the shorter, intermediate results, making it less efficient (although more correct) than \textit{isConnected}.

### 7.3 Incomplete Paths

There is a neater specification of paths that dispenses with the idea of calculating complete paths and then calculating subpaths. The subpaths can be directly specified via the following variation of \textit{perm} (5).

\[
\text{path}' &= \text{foldr maybeInclude} [] \\
\text{maybeInclude} &= \left( \lambda \; x \; xs \; \bullet \; \left( \begin{array}{c} \text{items} \; ys = \emptyset \\ \lor \\ \text{items} \; ys = \text{items} \; xs \end{array} \right) \right)
\]

That is, \textit{maybeInclude} is like \textit{include} (7), but doesn’t necessarily include the new element. Refinements of \textit{maybeInclude} follow a similar pattern to those in Sect. 3.1.

However, if this model is used, it is more difficult to specify the ‘invariant’. We hope to give this more consideration in the future.

### 7.4 Mindboggling Conclusions?

We haven’t yet done enough work on this example to decide whether we can take a neat specification of the problem involving conjunction, and perform a neat refinement that yields a neat result! However, the work that we’ve done so far does raise some interesting points.

### 8 Conclusions

There are examples where conjunction can be exploited to modularise the design of programs. This is particularly true when higher-order functions are used to build identical frameworks for the conjuncts, allowing a promotion-like mechanism to be used. The approach is unlikely to be suitable for implementing all programs that can be specified using conjunction and catamorphisms, but when it can be used it is quite elegant. In particular, our method of implementing an arbitrary restriction using \textit{foldr} can introduce concerns about monotonicity; the problem is similar to choosing a loop invariant that is too strong. The approach is not universal, but it does show some promise.

Another conclusion that can be drawn from this work is that problems can often be solved in elegant ways by combining elegant techniques from a variety of elegant methodologies.
References

10. B. P. Mahony. Expression refinement in higher order logic. In Grundy et al. [7], pages 230–249.


Using the Object Calculus to build an Object-Oriented Refinement Calculus

Jamie Shield and David Carrington
Department of Computer Science and Electrical Engineering
University of Queensland

Abstract. With the increasing maturation of object calculi, the investigation of object-oriented language foundations becomes easier. Object calculi can be used to detail many features of both class-based and object-based object-oriented languages. There is increasing interest in the development of object-oriented languages with Hoare style logics, or in a similar vein, object-oriented refinement calculi. This work in progress details some viable and non-viable options for the development of an object-oriented refinement calculus, using an object calculus, within a predicate transformer framework. This should aid refiners familiar with Back or Morgan style refinement calculi to with the use of the new object-oriented refinement calculi. The object-oriented language developed will feature the refinement of classes, abstract data types, and parametric polymorphic (generic) classes. While the development is not complete, it is hoped that this paper will spark questions and feedback about the approaches followed.

1 Introduction

Object orientation[3] is a programming paradigm that exploits analogies between encapsulations of code and data, termed objects, and real-life entities. Recent research on the theoretical foundations of object-oriented languages has led to the development of object calculi[1] which take objects as primitive. The object calculi can be used as the basis for both class-based and object-based languages. The refinement calculus[4] is a notation and set of rules developed to allow programs to be derived from specifications. Object-oriented refinement calculi are similar, except object-oriented programs, rather than procedural programs, are derived.

One general approach for developing an object-oriented refinement calculus built on an object calculus is to embed the object calculus into a existing predicate transformer based refinement calculus. Selerinski’s existentially typed refinement calculus[6] is an example of a predicate transformer based refinement calculus to which objects encoded using records have been added. Various techniques for developing such object-oriented refinement calculi are discussed in this paper and comparisons made on aspects such as simplicity and range of support. The paper explores different encodings of objects.
2 Object Calculus

Lambda calculi declare functions as primitive and provide syntax for manipulating the represented functions. Object calculi take objects as primitive. Research has shown how object-based languages [7] can be used to simplify and generalise the concepts in class-based languages. This is achieved by isolating the concepts of classes into more primitive entities. Object calculi are a further generalisation of object-based languages [1, p2]. The constructs in object calculi can be used to construct class-based language concepts, such as classes.

It is our hypothesis that defining object-oriented refinement calculi in terms of object calculi would produce a more intuitive, more cohesive semantics than definition in terms of records, functions and lambda calculi. The definition of object-oriented refinement calculi using lambda calculi has proven to be troublesome. Indeed, many developments require complex packaging of records and/or functions as well as extensions of object-oriented concepts such as subtyping [6, 5]. The object calculi, on the other hand, was developed with object-oriented features in mind. The development of an object-oriented refinement calculus should consequently be a more natural extension if based on an object calculus rather than an extended lambda calculus.

There are many examples of object calculi such as [2] and those summarised in [1]. An example object calculus, $FOb_{<\mu}$, from [1] has the following syntactic constructs. It is a functional object calculus with recursive types.

**Method declaration** $\varsigma(x) \text{ body}$

where $\varsigma$ is the object quantifier by analogy with $\lambda$ in the lambda calculus.

The term $x$ is the self parameter which is bound to the host object. It can be referenced within the body of the method declaration.

**Object declaration** $[l_1 = \varsigma(x_1) \text{ body}_1, ..., l_n = \varsigma(x_n) \text{ body}_n]$

Object declarations are collections of labels and associated methods. The containing object of a method is termed its host object. The order of the methods is irrelevant and methods that do not reference their self parameters can be termed fields.

The $FOb_{<\mu}$ calculus is typed. The typing system is used to introduce the notion of subtyping. The subtyping relation that is introduced, namely $<\mu$, is syntactic.

An object of type $[l_i : B_i^{i \in 1..n}]$ is a collection of methods with self parameters and whose bodies have types $B_1, ..., B_n$. For instance,

$$[\text{mem} = \varsigma(x) \ 5] : [\text{mem} : \text{Int}]$$

The subtyping rule for objects allows objects to be subtypes of objects with a subset of the attributes of the former object.

(l_i distinct)

$$\forall i \in 1..n + m \bullet (E \vdash B_i)$$

$$E \vdash [l_i : B_i^{i \in 1..n+m}] <\mu [l_i : B_i^{i \in 1..n}]$$
where \( \forall i \in \ldots n + m \bullet (E \vdash B_i) \) is actually \( n + m \) premises, namely \( E \vdash B_1 \) and \( E \vdash B_2 \) etc. The construct \( [l_i : B_i^{1..n}] \) is an object with \( n \) attributes, namely \( l_i : B_1 \), and \( l_2 : B_2 \) etc.

**Method invocation** With \( o = [l_i = \varsigma(x_i) b_i^{1..n}] \) and \( l_i \) distinct,

\[
o \cdot l_j \mapsto b_j[\{x_j \leftarrow o\}]
\]

where \( j \in \ldots n \) and \( \mapsto \) is object reduction. With \( o \) bound to the self parameter, \( x_j \) the method named \( l_j \) is invoked and the result of the execution is returned. The binding of the self parameter is effected by syntactic substitution.

**Method update** Again, with \( o = [l_i = \varsigma(x_i) b_i^{1..n}] \) and \( l_i \) distinct,

\[
o \cdot l \mapsto \varsigma(x) b \mapsto [l_j = \varsigma(y) b, l_i = \varsigma(x_i) b_i^{1..n} \setminus \{j\}]
\]

where \( j \in \ldots n \). A copy of \( o \) is made with the method \( l \) replaced with \( \varsigma(x) \) body. The new modified object is returned.

As an example, the following object, \( o_1 \), consists of a field \( \text{mem} \) of value 5 and a method \( \text{increment} \) which increments the field \( \text{mem} \).

\[
o_1 \doteq [\text{mem} = \varsigma(x) 5, \text{increment} = \varsigma(x) x.\text{mem} \mapsto \varsigma(y) x.\text{mem} + 1]
\]

The \( \mapsto \) symbol is the symbol for object reduction. The \( \{\} \) brackets denote syntactic substitution. When the \( \text{mem} \) field is selected, the result is the body of the \( \text{mem} \) field with all occurrences of the self parameter \( x \) replaced with (bound to) the host object. Since \( \text{mem} \) is a field, the body of \( \text{mem} \) does not contain any occurrences of the self parameter.

\[
o_{1.\text{mem}}
\]
\[
= [\text{byfield}^* o_1]
\]
\[
[\text{mem} = \varsigma(x) 5, \text{increment} = \varsigma(x) x.\text{mem} \mapsto \varsigma(y) x.\text{mem} + 1].\text{mem}
\]
\[
\mapsto [\text{byfieldselecton}]
\]
\[
5 \{\{x \leftarrow \text{mem} = \varsigma(x) 5,
\text{increment} = \varsigma(x) x.\text{mem} \mapsto \varsigma(y) x.\text{mem} + 1\}\}
\]
\[
= [\text{nullsubstitution}]
\]
\[
5
\]

Invoking the increment operation returns (reduces to) the body of the increment method with the method’s self parameter, \( x \), bound to (or substituted with) the self or host object.

\[
[\text{mem} = \varsigma(x) 5, \text{increment} = \varsigma(x) x.\text{mem} \mapsto \varsigma(y) x.\text{mem} + 1].\text{increment}
\]
\[
\mapsto [\text{bymethodinvocation}]
\]
\[
x.\text{mem} \mapsto \varsigma(y) x.\text{mem} + 1 \{\{x \leftarrow z\}\}
\]

where

\[
z = [\text{mem} = \varsigma(x) 5, \text{increment} = \varsigma(x) x.\text{mem} \mapsto \varsigma(v) x.\text{mem} + 1]
\]
The invocation of the increment method reduces to the body of the increment method, \(x.\text{mem} \leftarrow \zeta(y) x.\text{mem} + 1\), with \(x\) replaced with the self object. The self parameter in the substitution was renamed from \(y\) to \(v\) to avoid variable capturing in the resulting substitution. Naturally, it must be ensured that \(v\) does not occur in the object being substituted. By replacing both \(x\)'s, this is equivalent to:

\[
\begin{align*}
mem &= \zeta(x) 5, \\
\text{increment} &= \zeta(x) x.\text{mem} \leftarrow \zeta(v) x.\text{mem} + 1.\text{mem} \\
\end{align*}
\]

To effect a method update, the body of method within the host is simply replaced with the new method body.

\[
\Rightarrow \text{[by method update]} \begin{align*}
mem &= \zeta(y) \ [mem &= \zeta(x) 5, \\
\text{increment} &= \zeta(x) x.\text{mem} \leftarrow \zeta(v) x.\text{mem} + 1.\text{mem} + 1, \\
\text{increment} &= \zeta(x) x.\text{mem} \leftarrow \zeta(v) x.\text{mem} + 1.\text{mem} + 1 \end{align*}
\]

Notice that the evaluation of

\[
\begin{align*}
mem &= \zeta(x) 5, \\
\text{increment} &= \zeta(x) x.\text{mem} \leftarrow \zeta(v) x.\text{mem} + 1.\text{mem} + 1
\end{align*}
\]

has not been performed. It is not until the \(mem\) field is selected that this evaluation takes place. When a method is invoked (or a field is selected), the result is the body of the method selected with the self parameter bound to the host object.

\[
o_1.\text{increment}.\text{mem} = \text{[as shown above]} \begin{align*}
mem &= \zeta(y) \ [mem &= \zeta(x) 5, \\
\text{increment} &= \zeta(x) x.\text{mem} \leftarrow \zeta(v) x.\text{mem} + 1.\text{mem} + 1, \\
\text{increment} &= \zeta(x) x.\text{mem} \leftarrow \zeta(v) x.\text{mem} + 1.\text{mem} + 1 \end{align*}
\]

\[
\Rightarrow \text{[by field selection]} \begin{align*}
mem &= \zeta(x) 5, \\
\text{increment} &= \zeta(x) x.\text{mem} \leftarrow \zeta(v) x.\text{mem} + 1.\text{mem} + 1 \end{align*}
\]

where

\[
\begin{align*}
\text{increment} &= \zeta(x) x.\text{mem} \leftarrow \zeta(v) x.\text{mem} + 1.\text{mem} + 1, \\
\end{align*}
\]

\[
\begin{align*}
\text{increment} &= \zeta(x) x.\text{mem} \leftarrow \zeta(v) x.\text{mem} + 1
\end{align*}
\]

with appropriate self variable renamings to avoid variable capture.
\[
(2.1)
\]
\[
= \text{nullsubstitution}
\]
\[
\text{mem} = \varsigma(x) 5,
\]
\[
\text{increment} = \varsigma(x) x. \text{mem} \vdash \varsigma(v) x. \text{mem} + 1
\]
\[
\implies \text{hyfieldselection}
\]
\[
5 + 1 = \text{nullsubstitution}
\]
\[
5 + 1 = 6
\]

**Functions** The object calculus can be used to encode the lambda calculus. The lambda calculus syntax is used to abbreviate the actual objects they represent.

**Function abstraction**

\[
\lambda(x : A)b
\]

**Function application**

\[
b(a)
\]

For example:

\[
o_3 = \text{nullsubstitution}
\]
\[
\text{mem} = \varsigma(x) 5, \ldots, \text{increment} = \varsigma(x) x. \text{mem} \vdash \varsigma(y) (\lambda(z) z + 1)(x. \text{mem})
\]

The function \((\lambda(z : \text{Int})z + 1)\) takes an integer and returns its successor. Its application to \(x. \text{mem}\),

\[
(\lambda(z) z + 1)(x. \text{mem})
\]

returns the successor to \(x. \text{mem}\), provided that \(x. \text{mem}\) is an integer. The type of functions is:

\[
A \rightarrow B
\]

For instance, \((\lambda(a : \text{Int})a + 1)\) is of type \((\text{Int} \rightarrow \text{Int})\).

**Recursion** With the previous constructs, it is possible to type objects that use self or modify self and return a component, but it is not possible to type an object that returns self or a modified version. Recursive types provide this facility.

\[
\mu(X)B\{X\}
\]

is the unique solution to \(X = B\{X\}\) where \(B\{X\}\) means that \(X\) can occur freely within \(B\). The types \(\mu(X)B\{X\}\) and \(B\mu(X)B\{X\}\) are not equal. Some researchers equate these types thereby removing the need for fold and unfold constructs. For technical reasons, Abadi and Cardelli identify
these types as isomorphic and use \textit{fold} and \textit{unfold} constructs to relate these isomorphisms. 

The recursive type $A \doteq \mu(X)[l : X]$ is an object with a attribute $l$ that is recursive. Given an element of that type, $a : A$, unfolding $a$ will lead to an object with a type equivalent to the body of the type substituted with the type:

\[ \text{unfold}(a) : [l : A] \]

This allows access to the attributes at the first level. Unfolding twice provides access to both the first and second level attributes as it is typed as such:

\[ \text{unfold} \left( \text{unfold}(a) \right) : [l : [l : A]] \]

Refolding the singularly unfolded element leads to the original type.

\[ \text{fold}(A, \text{unfold}(a)) : A \]

Since $A$ and $[l : A]$ are different types, we do NOT have that $a$ is an element of $[l : A]$.

Since the increment operation returns a modified self, it requires a recursive type.

\[ o_1 \doteq [\text{mem} = \varsigma(x) \ 5, \ \text{increment} = \varsigma(x) \ 0, \ \text{mem} = \varsigma(y) \ 0, \ \text{mem} = \text{mem} + 1] : \mu(X)[\text{mem} \ : \ \text{Int}, \ \text{increment} \ : \ X] \]

\textbf{Existential Quantification} To allow data abstraction, existential types are provided.

Given the subtyping relation, $<:\prime$, the type $\exists(X <: A)B\{X\}$ is an existential type where $X$ is the quantified variable which must be a subtype of $A$ and $X$ can occur freely within $B$. Given $A'$ which is a subtype of $A$, and $b : B\{A'\}$, the pair $<A', b>$ has type $\exists(X <: A)B\{X\}$.

For instance $\exists(X <: \text{Int})X \times (X \rightarrow X)$ is an object type which allows abstraction from the implementation of the pair—some data and a function that is applicable to the data, $X \times (X \rightarrow X)$. We know the type is a product type with the second element a function and we know it works on subtypes of integers. A natural number and the successor function of natural numbers can be wrapped up into such a partially abstract type.

\[ p \doteq < \text{Nat}, (7, \text{succNat}) > : \exists(X <: \text{Int})X \times (X \rightarrow X) \]

To use the function on the data, both the data and the function need to be extracted. This is achieved with the \textit{open} construct.

\[ \text{open} \ e \ X <: A, x : B\{X\} \ \text{in} \ d\{X, x\} : D \]

Given $c : \exists(X <: A)B\{X\}$, it is unpacked into $x$. The body of the statement, $d$, has knowledge of the representation type, $X$, and access to $x$. The body, $d$, must not contain code that depends on $X$. For example, 

\[ \text{open} \ p \ \text{as} \ X <: \text{Int}, x : X \times (X \rightarrow X) \ \text{in} \ (\text{snd}(x))(\text{fst}(x)) \]
which results in
\[
x : (X \times (X \to X)) = (7, \text{succ}_{\text{Nat}})
\]

Consequently,
\[
(snd(x))(fst(x)) = \text{succ}_{\text{Nat}}(7) : X = 8 : X
\]

Since it is known that \( X <: \text{Int} \), the following typings can be derived.

\[
\text{open } p \text{ as } X <: \text{Int}, x : X \times (X \to X) \text{ in } \\
(snd(x))(fst(x)) : \text{Int} = 8 : \text{Int}
\]

\section{Approaches}

Given the object calculus, the task is to produce an object-oriented refinement calculus using it. In general, we provide the syntax for an object-oriented refinement language and provide predicate transformer semantics for those constructs. Using a lambda calculus extended with existential types, Sekerinski developed an object-oriented refinement calculus. We utilise this development to provide our first object-oriented refinement calculus built using the object calculus. Since a lambda calculus with existential types is a subset of the object calculus we described above, it is a relatively trivial task to provide the mapping to the object calculus instead. Sekerinski used existential types containing records to represent objects. By hiding the fields, Sekerinski achieves data abstraction. His main aim, however, was to use the hiding mechanism to provide subtyping of his ‘objects’.

Rather than use complex encodings of records, the representation of objects as object calculus objects may lead to a more intuitive semantics. The object calculus models subtyping, inheritance, classes and many other object-oriented features in a straightforward manner using just the constructs described in the previous section. It would be beneficial to use the object calculus models of object-oriented features rather than those used by Sekerinski.

The semantics are provided by initially defining the syntactic constructs that form an object-oriented refinement language—a variant of the original refinement language (the guarded command language)—and subsequently providing a translation from the syntactic constructs to objects within the object calculus.

While the chosen object calculus is functional, the object-oriented refinement language is imperative. States are records and predicates are functions on these states. Predicate transformers are functions on predicates. Using function composition, the predicate transformers are composed into a program. Consequently, a program is an object which is an encoded function that accepts an object (an encoded predicate) representing a postcondition and returns the predicate representing the weakest precondition necessary to establish the postcondition.

The original approach explored was the provision of a direct link between the code inside object definitions within the language and object calculus constructs. That is, object definitions would be simple object calculus object definitions.
Similarly, object inheritance would be modelled as simple object calculus inheritance. Since each language command must actually be a predicate transformer, the object (or class) definition would associate a state label (reference) with the object. That is, the object language object assignment:

\[
\begin{align*}
o & := \text{object} \\
& \quad \text{mem} : \text{Int} := 6 \\
& \quad \text{method increment} \triangleq \text{mem} := \text{mem} + 1 \\
\end{align*}
\]

actually represents a predicate transformer that maps the state label \( o \) to the directly translated object calculus object. We represent the predicate transformer, informally, by an assignment to \( o \).

\[
o := [\text{mem} = 6, \text{increment} = \zeta(x) x.\text{mem} \leftarrow x.\text{mem} + 1]
\]

Additionally, method and field\(^1\) update would also be associated with an assignment. The object language syntax

\[
o.\text{increment}
\]

would become the object calculus command:

\[
o := o.\text{increment}
\]

The \( o.\text{increment} \) would return a modified self with the \( \text{mem} \) field appropriately updated. This modification is consequently recorded by the assignment. The assignment provides the additional facility of turning the object calculus construct into an appropriate predicate transformer.

Observant readers will begin to notice some of the problems associated with this approach. Perhaps the most obvious approach is that objects would be restricted to modifying themselves. Any alterations made to other objects not contained within the object originally invoked would not be recorded. In other words, only the modifications performed to the original object invoked are recorded by the assignment. There is no obvious solution to this problem. Perhaps one could devise a solution which meant that an object invocation would be an assignment to all references in the state. Those not modified would be assigned to themselves. Given the state: \([o = [\text{mem} = 6, \text{increment} = \ldots], p = 9, q = 3] \]

\[
o.\text{increment}
\]

could perhaps mean

\[
o, p, q := o.\text{increment}, p, q
\]

\(^1\) Although we mention fields, both fields and methods are treated equally. Fields are provided with several abbreviations that show they are simply methods that don’t use their self parameter.
Another problem with this approach is that object types cannot be used. Since the \textit{increment} method returns a version of itself, we face similar problems as those that motivated the introduction of recursive types. Such typing problems could be overcome by using an imperative version of the object calculus. That is, a method update or invocation would produce side effects that updated self and returned nothing. We have not explored the incorporation of an imperative object calculus into the predicate transformer semantics framework.

Another solution is the use of recursive types. For instance, the $o$ object could be typed as

$$
\mu(X)[\text{mem} : \text{Int}, \text{increment} : X]
$$

Unfortunately, in order to subtype objects with methods that return self, using recursive types, a restriction needs to be applied. That is, the object’s methods cannot be updated. If they are allowed to be updated, subsumption of an object leads to unsound behaviour in some circumstances. For instance, given recursive types

$$
P_1 \equiv \mu(X)[\text{mem} : \text{Int}, \text{set} : \text{Int} \to X, \text{setfive} : X] \\
P_2 \equiv \mu(X)[\text{mem} : \text{Int}, \text{set} : \text{Int} \to X, \text{setfive} : X, \text{decrement} : X]
$$

and, informally,

$$
p_1 : P_1 \equiv [\text{mem} = 5, \text{set} = \zeta(x) \lambda(y : \text{Int}) x. \text{mem} \leftarrow y, \\
\text{setfive} = \zeta(x) x. \text{mem} \leftarrow 5] \\
p_2 : P_2 \equiv [\text{mem} = 5, \text{set} = \zeta(x) \lambda(y : \text{Int}) x. \text{mem} \leftarrow y, \\
\text{setfive} = \zeta(x) (x. \text{set}(6)). \text{decrement}, \\
\text{decrement} = \zeta(x) x. \text{mem} \leftarrow x. \text{mem} - 1]
$$

If $P_2 <: P_1$ then $p_2 : P_1$. Thus if, informally,

$$
q : P_1 \equiv p_2. \text{set}(\lambda x : \text{Int} \bullet p_1)
$$

then

$$
q = [\text{mem} = 5, \text{set} = (\lambda x : \text{Int} \bullet p_1), \text{setfive} = \zeta(x) (x. \text{set}(6)). \text{decrement}, \\
\text{decrement} = \zeta(x) x. \text{mem} \leftarrow x. \text{mem} - 1]
$$

Consequently, $q.\text{set}(z)$ ignores the argument, $z$, and returns an object of type $P_1$ with \text{mem} set to 5. The object that is returned by $q.\text{set}(z)$ does not have a \text{decrement} attribute. The $q$ object, however, \textit{does} have a \text{decrement} method. Even though $q$ has type $P_1$, the \text{decrement} method is still present. To clarify further, $p_2 : P_2$ and hence, via $P_2 <: P_1, p_2 : P_1$. The object $p_2$ still possesses a \text{decrement} method. If it didn’t, then the \text{setfive} method wouldn’t be sound thereby leading to a contradiction in the soundness of both the recursive subtyping rule and the original subtyping
rule. Thus

\[
\begin{align*}
q.setfive \\
= (by \ \text{def}^b \ of \ q) \\
(p_2.set = (\lambda x : \text{Int} \bullet p_1)).setfive \\
\implies \\
[\text{mem} = 5, set = (\lambda x : \text{Int} \bullet p_1), setfive = \zeta(x) (x.set(6)).decrement, \\
\quad \text{decrement} = \ldots].setfive \\
\implies \\
([\text{mem} = 5, set = (\lambda x : \text{Int} \bullet p_1), setfive = \zeta(x) (x.set(6)).decrement, \\
\quad \text{decrement} = \ldots].set(6)).decrement \\
\implies \\
p_1.decrement
\end{align*}
\]

However, \(p_1\) does not have a \textit{decrement} attribute thereby contradicting \(P_1 < P_2\).

Additionally, due to technicalities with the encoding of class types, subtyping of classes and consequently inheritance becomes unsound. See [1, p124] for details regarding both this problem and the previous problem.

We also looked at splitting the object into two sections (as Sekerinski does). In this case, the methods require knowledge of the host (of the object in which the methods are hosted) in order to access the fields. This subverts the point of splitting the object into two sections.

\[
\begin{align*}
l_1 & \equiv [\text{fields} = [\text{mem} = 6], \text{methods} = \zeta(y) \\
& \quad [\text{increment} = \zeta(x) y.\text{fields} \triangleright (y.\text{fields.\text{mem}} \triangleright y.\text{fields.\text{mem}} + 1)]]
\end{align*}
\]

A more sophisticated approach to the splitting technique is to model the objects such that their methods need only return the fields.

\[
\begin{align*}
l_2 & \equiv [\text{fields} = [\text{mem} = 6], \\
& \quad \text{increment} = \zeta(x) x.\text{fields} \triangleright (x.\text{fields.\text{mem}} \triangleright x.\text{fields.\text{mem}} + 1)]
\end{align*}
\]

Thus the invocation of the \textit{increment} method returns a modified version of the \textit{fields} field. By using the object language syntax of a method invocation to represent an update of the \textit{fields} field, the need for the return of a modified self is removed.

\[
l_2.\text{increment} \triangleq l_2.\text{fields} \triangleright l_2.\text{increment}
\]

The addition of methods can be easily subtyped.

\[
\begin{align*}
[\text{fields} : [\text{mem} : \text{Int}], \text{increment} : [\text{mem} : \text{Int}]] \\
<: \\
[\text{fields} : [\text{mem} : \text{Int}], \text{increment} : [\text{mem} : \text{Int}], \text{decrement} : [\text{mem} : \text{Int}]]
\end{align*}
\]

Unfortunately, the addition of fields is not easily subtyped. Simple object types are subtypes of shorter object types. The attributes types, however, need
to be invariant.

\[\text{[fields} : [\text{mem} : \text{Int}], \text{increment} : [\text{mem} : \text{Int}]]\]

\[\not\subset:\]

\[\text{[fields} : [\text{mem} : \text{Int}, \text{mem2} : \text{Int}], \text{increment} : [\text{mem} : \text{Int}, \text{mem2} : \text{Int}]]\]

Consequently, a more advanced typing of objects is required if this approach is followed. Such a type would be covariant in its components. Record types are an example that possesses this property. The use of this splitting approach in conjunction with disjoint unions (or sum types) may also hold some promise [1, p122],[5].

The use of variance annotations also allow covariance in objects’ components. Variance annotations are used to provide flexible subtyping. The \(\sim\) annotation is contravariance which prevents invocation yet allows contravariant subtyping,

\[-B : -B' \text{ if } B' : B\]

The \(\circ\) annotation is standard invariance, while \(+\) is covariance which prevents update yet allows covariant subtyping,

\[-B : -B' \text{ if } B : B'\]

The use of variance annotations will not solve the problem arising from using the splitting approach. Although variance annotations allow the subtypings desired, they will not allow the \textit{fields} field to be updated.

\[\text{[fields}^+ : [\text{mem} : \text{Int}], \text{increment}^+ : [\text{mem} : \text{Int}]]\]

\[\not\subset:\]

\[\text{[fields}^+ : [\text{mem} : \text{Int}], \text{increment}^+ : [\text{mem} : \text{Int}], \text{decrement}^+ : [\text{mem} : \text{Int}]]\]

\[\subset:\]

\[\text{[fields}^+ : [\text{mem}, \text{mem2} : \text{Int}], \text{increment}^+ : [\text{mem}, \text{mem2} : \text{Int}]]\]

Finally, we explored the use of predicate transformers as the bodies of methods. Consequently, objects have types such as the following.

\[\text{m} \triangleq [\text{mem} = 6, \text{increment} = \varsigma(x) \ x.\text{mem} := x.\text{mem} + 1] : \]

\[\text{[mem} : \text{Int}, \text{increment} : \text{Trans} \ A \ B]\]

where \(A\) and \(B\) are the state spaces and the field update, \(x.\text{mem} := x.\text{mem} + 1\) is actually a predicate transformer that updates the host object.

\[\text{m} := \text{m.mem} \triangleq \text{m.mem} + 1\]

The assignment above should be treated as a predicate transformer. The syntax on the right hand side of the assignment is pure object calculus code once \(m\) is replaced with the value the state has stored for it.

Since a method invocation returns a predicate transformer, a method invocation in the object language is mapped directly to a method invocation in the
object calculus. The trouble with this approach is similar to the original one. That is, the method \textit{increment} modifies the host object of which it is a part. Consequently the host object must be a part of the state spaces \(A\) and \(B\). The solutions of recursion and self types have been explored.

One problem with the Self types approach is that contravariant occurrences of self are unsound. The way that Self types are defined enable them to be efficiently used in conjunction with subsumption. They do not, however, allow contravariant occurrences of the self parameter. Function types are covariant in their first parameter. The following type was not considered plausible for an encoding of objects as any occurrences of \(X\) inside \(B\) would occur in contravariant positions.

\[
\zeta(X)[mem : \text{Int}, \text{increment} : \text{Trans} A\{X\} \to B\{X\}] = \\
\zeta(X)[mem : \text{Int}, \text{increment} : B\{X\} \to A\{X\}]
\]

Recursive types, however, do allow contravariant occurrences of self.

\[
n : \mu(X)[mem : \text{Int}, \text{increment} : \text{Trans} A\{X\} \to B\{X\}]\]

However, with the type above, \(n : \mu(X)[\ldots]\) needs to be part of both \(A\) and \(B\) for it to type correctly. If it weren’t, the predicate transformer associated with the \textit{increment} method wouldn’t be able to alter \(n\). A similar object, for instance,

\[
p : \mu(X)[mem : \text{Int}, \text{increment} : \text{Trans} A\{X\} \to B\{X\}]\]

requires \(p : \mu(X)[\ldots]\) within \(A\) and \(B\). Consequently, two objects with the same implementation would require differing types.

A small modification allows this problem to be addressed. By passing any modified self object back separately, the state label that the object is associated with need no longer be part of the object’s type.

\[
\mu(X)[mem : \text{Int}, \text{increment} : (\text{Trans} A B) \times X]\]

The most promising type uses self types in a similar fashion. The problem of contravariant occurrences that were present in the previous example of self types encodings is avoided. Additionally, by using self types rather than recursive types, there is an increase in the subtyping capabilities.

\[
\zeta(X)[mem : \text{Int}, \text{increment} : (\text{Trans} A B) \times X]\]

4 Conclusion

Object methods can perform alterations on the host object. This means that the object is somewhat recursive in nature. Consequently, it isn’t possible to use a functional object calculus’ object type as the encoding for object within the development of an object-oriented refinement calculus based on predicate transformers.
This paper has discussed various different encodings and detailed examples of both plausible and non-plausible encodings. The most viable option involves a self typed encoding of objects. This is perhaps not surprising given that a functional calculus needs to return modified versions of objects when in order for them to be useful.

Some alternatives approaches have also been identified, yet not discussed, in this paper. For instance, the use of imperative object calculi may be possible.

The intention of this paper is to spark questions and feedback about the approaches detailed.

References

\(\tau\)-Equivalences and Refinement

Igor V. Tarasyuk

A.P. Ershov Institute of Informatics Systems
Lavrentieva ave. 6, 630090, Novosibirsk, Russia
Fax: +7 3832 32 34 94
itar@iis.nsk.su

Abstract. The paper is devoted to the investigation of behavioural equivalences for Petri nets with silent transitions. Basic \(\tau\)-equivalences and back-forth \(\tau\)-bisimulation equivalences are supplemented by new ones, giving rise to complete set of equivalence notions in interleaving / true concurrency and linear / branching time semantics. Their interrelations are examined, and the preservation of all the equivalence notions by refinements is investigated.

1 Introduction

The notion of equivalence is central in any theory of systems. It allows to compare systems taking into account particular aspects of their behaviour.

Petri nets became a popular formal model for design of concurrent and distributed systems. One of the main advantages of Petri nets is their ability for structural characterization of three fundamental features of concurrent computations: causality, nondeterminism and concurrency.

Silent transitions are transitions labelled by special silent action \(\tau\) which represents an internal activity of a system to be modelled and it is invisible for an external observer.

Equivalences which abstract of silent actions are called \(\tau\)-equivalences (these are labelled by the symbol ‘\(\tau\)’ to distinguish them of relations not abstracting of silent actions). The following basic notions of \(\tau\)-equivalences are known from the literature.

- \(\tau\)-trace equivalences (they respect only protocols of behaviour of systems): interleaving \((\equiv^\tau_{\text{tr}})\) [7], step \((\equiv^\tau_{\text{step}})\) [7], partial word \((\equiv^\tau_{\text{pw}})\) [10] and pomset \((\equiv^\tau_{\text{pom}})\) [7].

- Usual \(\tau\)-bisimulation equivalences (they respect branching structure of behaviour of systems): interleaving \((\leftrightarrow^\tau)\) [7], step \((\leftrightarrow^\tau)\) [7], partial word \((\leftrightarrow^\tau_{\text{pw}})\) [10] and pomset \((\leftrightarrow^\tau_{\text{pom}})\) [7].

- \(ST\)-\(\tau\)-bisimulation equivalences (they respect the duration or maximality of events in behaviour of systems): interleaving \((\leftrightarrow^\tau_{\text{ST}})\) [10], partial word \((\leftrightarrow^\tau_{\text{pwST}})\) [10] and pomset \((\leftrightarrow^\tau_{\text{pomST}})\) [10].

* The work is supported by INTAS-RFBR, grant 95-0378, Volkswagen Fund, grant 1/70 564 and the Foundation for Promotion to Young Scientists of Siberian Division of the Russian Academy of Sciences
- **History preserving \( \tau \)-bisimulation equivalences** (they respect the “past” or “history” of behaviour of systems): pomset \( \equiv_{\text{pom}} \) [7].

- **History preserving \( ST \)-\( \tau \)-bisimulation equivalences** (they respect the “history” and the duration or maximality of events in behaviour of systems): pomset \( \equiv_{\text{pomST}} \) [7].

- **Usual branching \( \tau \)-bisimulation equivalences** (they respect branching structure of behaviour of systems taking a special care for silent actions): interleaving \( \equiv_{\text{bbr}} \) [7].

- **History preserving branching \( \tau \)-bisimulation equivalences** (they respect “history” and branching structure of behaviour of systems taking a special care for silent actions): pomset history preserving \( \equiv_{\text{pombr}} \) [7].

- **Isomorphism** (\( \simeq \)) (i.e. coincidence of systems up to renaming of their components).

Another important group of equivalences are back-forth bisimulation ones which are based on the idea that bisimulation relation do not only require systems to simulate each other behaviour in the forward direction but also when going back in history. They are closely connected with equivalences of logics with past modalities.

These equivalence notions were initially introduced in [6]. On transition systems with silent actions it was shown that back-forth variant \( \equiv_{\text{bif}} \) of interleaving \( \tau \)-bisimulation equivalence coincide with \( \equiv_{\text{bbr}} \).

In [4] the new variants of step, partial word and pomset back-forth bisimulation equivalences were defined in the framework of event structures without silent actions.

In [8] the new idea of differentiating the kinds of back and forth simulations appeared. The set of all possible back-forth equivalence notions was proposed in interleaving, step, partial word and pomset semantics for event structures without silent actions. The new notion of \( \tau \)-equivalence was proposed for event structures with silent actions: pomset back pomset forth \( \equiv_{\text{pompom}} \) \( \tau \)-bisimulation equivalence. Its coincidence with \( \equiv_{\text{pombr}} \) was proved.

To choose most appropriate behavioural viewpoint on systems to be modelled, it is very important to have a complete set of equivalence notions in all semantics and understand their interrelations. This branch of research is usually called **comparative concurrency semantics**. Treating equivalences for preservation by refinements allows one to decide which of them may be used for top-down design.

Working in the framework of Petri nets with silent transitions, in this paper we continue research of [9] and extend the set of basic notions of \( \tau \)-equivalences by interleaving \( ST \)-branching \( \tau \)-bisimulation one \( \equiv_{\text{STbr}} \), pomset history preserving \( ST \)-branching \( \tau \)-bisimulation one \( \equiv_{\text{pomSTbr}} \) and multi-event structure one \( \equiv_{\text{mST}} \). Let us note that an idea to introduce \( \equiv_{\text{pomSTbr}} \) appeared initially in [8]. We complete back-forth \( \tau \)-equivalences from [8] by 6 new notions in interleaving - pomset semantics. We compare all the \( \tau \)-equivalences and obtain a diagram of their interrelations.
In [3], SM-refinement operator for Petri nets was proposed, which “replaces” their transitions by SM-nets, a special subclass of state machine nets. We treat all the \( \tau \)-equivalences for preservation by SM-refinements. We show that \( \tilde{\epsilon}_p \supseteq \tilde{\epsilon}_p \forall \tau \forall \tau \) and \( \equiv \tau \), i.e. all the new equivalences introduced in this paper, are preserved by SM-refinements. Thus, we have branching and conflict preserving equivalences which may be used for multilevel design. In the literature, a stability w.r.t. SM-refinements was proved only for \( \tilde{\epsilon}_p \supseteq \tilde{\epsilon}_p \forall \tau \) in [3] and for \( \tilde{\epsilon}_p \supseteq \tilde{\epsilon}_p \) in [5]. The preservation result for other \( \tilde{\epsilon}_p \supseteq \tilde{\epsilon}_p \) was proved in [10], but it was done on event structures and an other refinement operator was used. The preservation of trace \( \tau \)-equivalences was not established before. Thus, our results for \( \tilde{\epsilon}_p \supseteq \tilde{\epsilon}_p \) and \( \equiv \tau \) are also new.

2 Basic Definitions

In this section we give some basic definitions used further.

2.1 Labelled Nets

Let \( \text{Act} = \{a, b, \ldots \} \) be a set of action names or labels. The symbol \( \tau \not\in \text{Act} \) denotes a special silent action. We denote \( \text{Act}_\tau = \text{Act} \cup \{\tau\} \).

**Definition 1.** A labelled net is a quadruple \( N = \langle P_N, T_N, F_N, l_N \rangle \), where:

- \( P_N = \{p, q, \ldots \} \) is a set of places;
- \( T_N = \{t, u, \ldots \} \) is a set of transitions;
- \( F_N : (P_N \times T_N) \cup (T_N \times P_N) \rightarrow \textbf{N} \) is the flow relation with weights (\( \textbf{N} \) denotes a set of natural numbers);
- \( l_N : T_N \rightarrow \text{Act}_\tau \) is a labelling of transitions with action names.

Given labelled nets \( N = \langle P_N, T_N, F_N, l_N \rangle \) and \( N' = \langle P_{N'}, T_{N'}, F_{N'}, l_{N'} \rangle \). A mapping \( \beta : P_N \cup T_N \rightarrow P_{N'} \cup T_{N'} \) is an isomorphism between \( N \) and \( N' \), denoted by \( \beta : N \cong N' \), if:

1. \( \beta \) is a bijection such that \( \beta(P_N) = P_{N'} \) and \( \beta(T_N) = T_{N'} \);
2. \( \forall p \in P_N \forall t \in T_N \) \( F_N(p, t) = F_{N'}(\beta(p), \beta(t)) \) and \( F_N(t, p) = F_{N'}(\beta(t), \beta(p)) \);
3. \( \forall t \in T_N \) \( l_N(t) = l_{N'}(\beta(t)) \).

Labelled nets \( N \) and \( N' \) are isomorphic, denoted by \( N \cong N' \), if \( \exists \beta : N \cong N' \).

Given a labelled net \( N \) and some transition \( t \in T_N \), the precondition and postcondition \( t \), denoted by \( \preceq t \) and \( \succeq t \) respectively, are the multisets defined in such a way: \( \preceq t(p) = F_N(p, t) \) and \( \succeq t(p) = F_N(t, p) \). Analogous definitions are introduced for places: \( \preceq p(t) = F_N(t, p) \) and \( \succeq p(t) = F_N(p, t) \). Let \( \preceq N = \{p \in P_N \mid \preceq p = \emptyset\} \) is a set of input places of \( N \) and \( \succeq N = \{p \in P_N \mid \succeq p = \emptyset\} \) is a set of output places of \( N \).

A labelled net \( N \) is acyclic, if there exist no transitions \( t_0, \ldots, t_n \in T_N \) such that \( \preceq t_0 \cap \succeq t_i \neq \emptyset \) \((1 \leq i \leq n)\) and \( t_0 = t_n \). A labelled net \( N \) is ordinary if \( \forall p \in P_N \) \( \preceq p \) and \( \succeq p \) are proper sets (not multisets).

Let \( N = \langle P_N, T_N, F_N, l_N \rangle \) be acyclic ordinary labelled net and \( x, y \in P_N \cup T_N \). Let us introduce the following notions.
$x \prec_N y \iff x F^+_N y$, where $F^+_N$ is a transitive closure of $F_N$ (strict causal dependence relation);

$y \sim_N x \iff (x \prec_N y) \lor (x = y)$ (a relation of causal dependence);

$x \#_N y \iff \exists t, u \in T_N (t \neq u, \bullet_t \cap \bullet_u \neq \emptyset, t \preceq_N x, u \succeq_N y)$ (a relation of conflict);

$\downarrow_N x = \{ y \in P_N \cup T_N \mid y \approx_N x \}$ (the set of strict predecessors of $x$).

A set $T \subseteq T_N$ is left-closed in $N$, if $\forall t \in T \ (\downarrow_N t) \cap T_N \subseteq T$.

### 2.2 Marked Nets

We denote the set of all finite multisets over a set $X$ by $\mathcal{M}(X)$. A marking of a labelled net $N$ is a multiset $M \in \mathcal{M}(P_N)$.

**Definition 2.** A marked net (net) is a tuple $N = \langle P_N, T_N, F_N, l_N, M_N \rangle$, where $\langle P_N, T_N, F_N, l_N \rangle$ is a labelled net and $M_N \in \mathcal{M}(P_N)$ is the initial marking.

Given nets $N = \langle P_N, T_N, F_N, l_N, M_N \rangle$ and $N' = \langle P_{N'}, T_{N'}, F_{N'}, l_{N'}, M_{N'} \rangle$. A mapping $\beta : P_N \cup T_N \rightarrow P_{N'} \cup T_{N'}$ is an isomorphism between $N$ and $N'$, denoted by $\beta : N \simeq N'$, if $\beta : \langle P_N, T_N, F_N, l_N \rangle \simeq \langle P_{N'}, T_{N'}, F_{N'}, l_{N'} \rangle$ and $\forall p \in P_N \ M_N(p) = M_{N'}(\beta(p))$. Nets $N$ and $N'$ are isomorphic, denoted by $N \simeq N'$, if $\exists \beta : N \simeq N'$.

Let $M \in \mathcal{M}(P_N)$ be a marking of a net $N$. A transition $t \in T_N$ is fiirable in $M$, if $\bullet t \subseteq M$. If $t$ is fiirable in $M$, firing it yields a new marking $\hat{M} = M - \bullet t + \bullet^*$, denoted by $M \xrightarrow{t} \hat{M}$. $\text{Mark}(N)$ denotes a set of all reachable markings of a net $N$.

### 2.3 Partially Ordered Sets

**Definition 3.** A labelled partially ordered set (lposet) is a triple $\rho = \langle X, \prec, l \rangle$, where:

- $X = \{ x, y, \ldots \}$ is some set;
- $\prec \subseteq X \times X$ is a strict partial order (irreflexive transitive relation) over $X$;
- $l : X \rightarrow \text{Act}_\tau$ is a labelling function.

Let $\rho = \langle X, \prec, l \rangle$ be lposet and $Y \subseteq X$. A restriction of $\rho$ to the set $Y$ is defined as follows: $\rho|_Y = \langle Y, \prec \cap (Y \times Y), l|_Y \rangle$.

Let $\rho = \langle X, \prec, l \rangle$ and $\rho' = \langle X', \prec', l' \rangle$ be lposets.

A mapping $\beta : X \rightarrow X'$ is a label-preserving bijection between $\rho$ and $\rho'$, denoted by $\beta : \rho \simeq \rho'$, if $\beta$ is a bijection s.t. $\forall x \in X \ l(x) = l'(\beta(x))$. We write $\rho \simeq \rho'$, if $\exists \beta : \rho \simeq \rho'$.

A mapping $\beta : X \rightarrow X'$ is a homomorphism between $\rho$ and $\rho'$, denoted by $\beta : \rho \subseteq \rho'$, if $\beta : \rho \preceq \rho'$ and $\forall x, y \in X \ x \prec y \Rightarrow \beta(x) \prec' \beta(y)$. We write $\rho \subseteq \rho'$, if $\exists \beta : \rho \subseteq \rho'$.

A mapping $\beta : X \rightarrow X'$ is an isomorphism between $\rho$ and $\rho'$, denoted by $\beta : \rho \simeq \rho'$, if $\beta : \rho \subseteq \rho'$ and $\beta^{-1} : \rho' \subseteq \rho$. Lposets $\rho$ and $\rho'$ are isomorphic, denoted by $\rho \simeq \rho'$, if $\exists \beta : \rho \simeq \rho'$.
Definition 4. Partially ordered multiset (pomset) is an isomorphism class of \( \mathcal{L} \)-posets.

2.4 Event Structures

Definition 5. A labelled event structure (LES) is a quadruple \( \xi = \langle X, \prec, \#, l \rangle \), where:

- \( X = \{ x, y, \ldots \} \) is a set of events;
- \( \prec \subseteq X \times X \) is a strict partial order, a causal dependence relation, satisfying to the principle of finite causes: \( \forall x \in X \downarrow \exists x < \infty \);
- \( \# \subseteq X \times X \) is an irreflexive symmetrical conflict relation, satisfying to the principle of conflict heredity: \( \forall x, y, z \in X \; x \# y \wedge y \# z \Rightarrow x \# z \);
- \( l : X \to \text{Act}_r \) is a labelling function.

Let \( \xi = \langle X, \prec, \#, l \rangle \) be LES and \( Y \subseteq X \). A restriction of \( \xi \) to the set \( Y \) is defined as follows: \( \xi|_Y = \langle Y, \prec \cap (Y \times Y), \# \cap (Y \times Y), l|_Y \rangle \).

Let \( \xi = \langle X, \prec, \#, l \rangle \) and \( \xi' = \langle X', \prec', \#, l' \rangle \) be LES's. A mapping \( \beta : X \to X' \) is an isomorphism between \( \xi \) and \( \xi' \), denoted by \( \beta : \xi \simeq \xi' \), if:

1. \( \beta \) is a bijection;
2. \( \forall x \in X \; l(x) = l'(\beta(x)) \);
3. \( \forall x, y \in X \; x \prec y \Leftrightarrow \beta(x) \prec \beta(y) \);
4. \( \forall x, y \in X \; x \# y \Leftrightarrow \beta(x) \# \beta(y) \).

LES's \( \xi \) and \( \xi' \) are isomorphic, denoted by \( \xi \simeq \xi' \), if \( \exists \beta : \xi \simeq \xi' \).

Definition 6. A multi-event structure (MES) is an isomorphism class of LES's.

2.5 C-processes

Definition 7. A causal net is an acyclic ordinary labelled net \( C = \langle P_C, T_C, F_C, l_C \rangle \), s.t.:

1. \( \forall r \in P_C \; |r| \leq 1 \) and \( |r^*| \leq 1 \), i.e. places are unbranched;
2. \( \forall x \in P_C \cap T_C \; |\downarrow x| < \infty \), i.e. a set of causes is finite.

On the basis of any causal net \( C = \langle P_C, T_C, F_C, l_C \rangle \) one can define lpcset \( p_C = \langle T_C, \prec_C \cap (T_C \times T_C), l_C \rangle \).

The fundamental property of causal nets is: if \( C \) is a causal net, then there exists a sequence of transition firings \( ^*C = L_0 \xrightarrow{\xi_0} \cdots \xrightarrow{\xi_n} L_n = C^* \) such that \( L_i \subseteq P_C \; (0 \leq i \leq n) \), \( P_C = \bigcup_{i=0}^n L_i \) and \( T_C = \{ v_1, \ldots, v_n \} \). Such a sequence is called a full execution of \( C \).

Definition 8. Given a net \( N \) and a causal net \( C \). A mapping \( \varphi : P_C \cup T_C \to P_N \cup T_N \) is an embedding \( C \) into \( N \), denoted by \( \varphi : C \to N \), if:

1. \( \varphi(P_C) \in \mathcal{M}(P_N) \) and \( \varphi(T_C) \in \mathcal{M}(T_N) \), i.e. sorts are preserved;
2. \( \forall v \in T_C \; \varphi(v) = \varphi^*(v) \) and \( \varphi(v)^* = \varphi^*(v)^* \), i.e. flow relation is respected;
3. \( \forall v \in TC \ l_C(v) = l_N(\varphi(v)) \), i.e. labelling is preserved.

Since embeddings respect the flow relation, if \( ^C C \xrightarrow{\varphi(v_1)} \cdots \xrightarrow{\varphi(v_n)} C^o \) is a full execution of \( C \), then \( M = \varphi(C) \xrightarrow{\varphi(v_1)} \cdots \xrightarrow{\varphi(v_n)} \varphi(C^o) = \tilde{M} \) is an sequence of transition firings in \( N \).

**Definition 9.** A firable in marking \( M \) \( C \)-process (process) of a net \( N \) is a pair \( \pi = (C, \varphi) \), where \( C \) is a causal net and \( \varphi : C \to N \) is an embedding such that \( M = \varphi(C^o) \). A firable in \( M_N \) process is a process of \( N \).

We write \( \Pi(N, M) \) for a set of all firable in marking \( M \) processes of a net \( N \) and \( \Pi(N) \) for the set of all processes of a net \( N \). The initial process of a net \( N \) is \( \pi_N = (C_N, \varphi_N) \in \Pi(N) \), such that \( T_{C_N} = \emptyset \). If \( \pi \in \Pi(N, M) \), then firing of this process transforms a marking \( M \) into \( \tilde{M} = M - \varphi(C^o) + \varphi(C^o) = \varphi(C^o) \), denoted by \( M \xrightarrow{\pi} \tilde{M} \).

Let \( \pi = (C, \varphi) \), \( \hat{\pi} = (\hat{C}, \hat{\varphi}) \in \Pi(N) \), \( \hat{\pi} = (\hat{C}, \hat{\varphi}) \in \Pi(N, \varphi(C^o)) \). A process \( \hat{\pi} \) is an extension of \( \pi \) by process \( \hat{\pi} \), denoted by \( \hat{\pi} \xrightarrow{\pi} \hat{\pi} \), if \( T_C \subseteq T_{\hat{C}} \) is a left-closed set in \( \hat{C} \) and \( T_{\hat{C}} = T_C \setminus T_C \). We write \( \pi \to \hat{\pi} \), if \( \exists \bar{\pi} \pi \xrightarrow{\pi} \bar{\pi} \).

A process \( \bar{\pi} \) is an extension of a process \( \pi \) by one transition, denoted by \( \pi \xrightarrow{\varepsilon} \bar{\pi} \) or \( \pi \xrightarrow{v} \bar{\pi} \), if \( \pi \xrightarrow{v} \bar{\pi} \), \( T_{\hat{C}} = \{v\} \) and \( l_{\hat{C}}(v) = a \).

A process \( \bar{\pi} \) is an extension of a process \( \pi \) by sequence of transitions, denoted by \( \pi \xrightarrow{\pi_1} \pi_2 \cdots \pi_n \), if \( \exists \pi_i \in \Pi(N) (1 \leq i \leq n) \pi \xrightarrow{\pi_i} \pi_{i+1} \), \( \sigma = v_1 \cdots v_n \), and \( l_{\hat{C}}(\sigma) = \omega \).

A process \( \hat{\pi} \) is an extension of a process \( \pi \) by multiset of transitions, denoted by \( \pi \xrightarrow{\pi_1} \hat{\pi} \) or \( \pi \xrightarrow{\pi} \hat{\pi} \), if \( \pi \xrightarrow{\pi} \hat{\pi} \), \( \sigma = \emptyset \), \( T_{\hat{C}} = V \) and \( l_{\hat{C}}(V) = A \).

2.6 O-processes

**Definition 10.** An occurrence net is an acyclic ordinary labelled net \( O = \langle P_O, T_O, F_O, l_O \rangle \), s.t.:

1. \( \forall v \in P_O \ |*v| \leq 1 \), i.e. there are no backwards conflicts;
2. \( \forall v \in P_O \cup T_O -(x\#Ox) \), i.e. conflict relation is irreflexive;
3. \( \forall x \in P_O \cup T_O \ |\downarrow x| < \infty, \) i.e. set of causes is finite.

Let \( O = \langle P_O, T_O, F_O, l_O \rangle \) be occurrence net and \( N = \langle P_N, T_N, F_N, l_N, M_N \rangle \) be some net. A mapping \( \psi : P_O \cup T_O \to P_N \cup T_N \) is an embedding \( O \) into \( N \), notation \( \psi : O \to N \), if:

1. \( \psi(P_O) \in M(P_N) \) and \( \psi(T_O) \in M(T_N) \), i.e. sorts are preserved;
2. \( \forall v \in T_O \ l_O(v) = l_N(\psi(v)) \), i.e. labelling is preserved;
3. \( \forall v \in T_O \ \psi(v) = \psi(\psi(v)) \) and \( \psi(v)^* = \psi(\psi(v)^*) \), i.e. flow relation is respected;
4. \( \forall v, w \in T_O \ \psi(v) = \psi(w) \wedge (\psi(v) = \psi(w)) \Rightarrow v = w \), i.e. there are no "superfluous" conflicts.

**Definition 11.** An O-process of a net \( N \) is a pair \( \varpi = (O, \psi) \), where \( O \) is an occurrence net and \( \psi : O \to N \) is an embedding s.t. \( M_N = \psi(O) \).
We write $\mathcal{g}(N)$ for a set of all $O$-processes of a net $N$. The initial $O$-process of a net $N$ coincides with its initial $C$-process, i.e. $\omega_N = \pi_N$.

Let $\omega = (O, \psi), \omega = (O, \psi) \in \mathcal{g}(N), O = \langle P_O, T_O, F_O, l_O \rangle, \tilde{O} = \langle P_{\tilde{O}}, T_{\tilde{O}}, F_{\tilde{O}}, l_{\tilde{O}} \rangle$. An $O$-process $\tilde{\omega}$ is an extension of $\omega$, denoted by $\omega \rightarrow \tilde{\omega}$, if $T_O \subseteq T_{\tilde{O}}$ is a left-closed set in $\tilde{O}$.

An $O$-process $\omega$ of a net $N$ is maximal, if $\forall \omega = (O, \psi)$ s.t. $T_O \setminus T_O = \emptyset$. A set of all maximal $O$-processes of a net $N$ consists of the unique $O$-process $\omega_{\text{max}} = (O_{\text{max}}, \psi_{\text{max}})$. In such a case an isomorphism class of occurrence net $O_{\text{max}}$ is an unfolding of a net $N$, notation $\mathcal{U}(N)$.

Let us note that on the basis of any occurrence net $O$ one can define LES $\xi_O = \langle T_O, \sim_O \cap (T_O \times T_O), \#_O \cap (T_O \times T_O), l_O \rangle$. Then on the basis of unfolding $\mathcal{U}(N)$ of a net $N$ one can define MES $\mathcal{E}(N) = \xi_{\mathcal{U}(N)}$ which is an isomorphism class of LES $\xi_O$ for $O \in \mathcal{U}(N)$.

3 Basic $\tau$-equivalences

In this section we propose basic $\tau$-equivalences: trace, bisimulation and conflict preserving.

3.1 $\tau$-trace Equivalences

We denote the empty string by the symbol $\varepsilon$.

Let $\sigma = a_1 \cdots a_n \in \mathcal{A}^*$. We define $vis(\sigma)$ as follows ($a \in \mathcal{A}_{\tau}$).

1. $vis(\varepsilon) = \varepsilon$;
2. $vis(\sigma a) = \begin{cases} vis(\sigma)a, a \neq \tau; \\ vis(\sigma), a = \tau. \end{cases}$

Definition 12. A visible interleaving trace of a net $N$ is a sequence $\sigma_1 \cdots \sigma_n \in \mathcal{A}^*$ s.t. $\pi_N \xrightarrow{\sigma_1} \pi_2 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} \pi_n$, where $\pi_i \in \Pi(N)$ (1 $\leq i \leq n$). We denote a set of all visible interleaving traces of a net $N$ by VisInterTraces($N$).

Two nets $N$ and $N'$ are interleaving $\tau$-trace equivalent, denoted by $N \equiv_\tau^I N'$, if $\text{VisInterTraces}(N) = \text{VisInterTraces}(N')$.

Let $\Sigma = A_1 \cdots A_n \in (\mathcal{M}(\mathcal{A}_{\tau}))^*$. We define $vis(\Sigma)$ as follows ($A \in \mathcal{M}(\mathcal{A}_{\tau})$).

1. $vis(\varepsilon) = \varepsilon$;
2. $vis(\Sigma A) = \begin{cases} vis(\Sigma)(A \cap \mathcal{A}_{\tau}), A \cap \mathcal{A}_{\tau} \neq \emptyset; \\ vis(\Sigma), \text{ otherwise}. \end{cases}$

Definition 13. A visible step trace of a net $N$ is a sequence $vis(A_1 \cdots A_n) \in (\mathcal{M}(\mathcal{A}_{\tau}))^*$ s.t. $\pi_N \xrightarrow{A_1} \pi_1 \xrightarrow{A_2} \cdots \xrightarrow{A_n} \pi_n$, where $\pi_i \in \Pi(N)$ (1 $\leq i \leq n$). We denote a set of all visible step traces of a net $N$ by VisStepTraces($N$). Two nets $N$ and $N'$ are step $\tau$-trace equivalent, denoted by $N \equiv_\tau^S N'$, if $\text{VisStepTraces}(N) = \text{VisStepTraces}(N')$. 
Let $\rho = (X, -, l)$ be iposet s.t. $l : X \rightarrow Act_\tau$. We denote $vis(X) = \{x \in X \mid l(x) \in Act\}$ and $vis(\rho) = \rho |_{vis(X)}$.

**Definition 14.** A visible pomset trace of a net $N$ is a pomset $vis(\rho)$, an isomorphism class of iposet $vis(\rho_C)$ for $\pi = (C, \phi) \in \Pi(N)$. We denote a set of all visible pomsets of a net $N$ by $VisPomsets(N)$. Two nets $N$ and $N'$ are partial word $\tau$-trace equivalent, denoted by $N \equiv^\tau_{pw} N'$, if $VisPomsets(N) \subseteq VisPomsets(N')$ and $VisPomsets(N') \subseteq VisPomsets(N)$.

**Definition 15.** Two nets $N$ and $N'$ are pomset $\tau$-trace equivalent, denoted by $N \equiv^\tau_{pom} N'$, if $VisPomsets(N) = VisPomsets(N')$.

### 3.2 $\tau$-bisimulation Equivalences

Let $C = (P_C, T_C, F_C, I_C)$ be C-net. We denote $vis(T_C) = \{v \in T_C \mid I_C(v) \in Act\}$ and $vis(\prec_C) = \prec_C \cap (vis(T_C) \times vis(T_C))$.

#### Usual $\tau$-bisimulation Equivalences

**Definition 16.** Let $N$ and $N'$ be some nets. A relation $R \subseteq \Pi(N) \times \Pi(N')$ is a $\prec$-$\tau$-bisimulation between $N$ and $N'$, $\star \in \{\text{interleaving, step, partial word, pomset}\}$, denoted by $N \leftrightarrow^\tau R N'$, $\star \in \{i, s, pw, pom\}$, if:

1. $(\pi_N, \pi_{N'}) \in R$.
2. $(\pi, \pi') \in R$, $\pi \xrightarrow{\star} \tilde{\pi}$, $\tilde{\pi} 

   (a) $|vis(T_{C})| = 1$, if $\star = i$;
   (b) $vis(\prec_C) = \emptyset$, if $\star = s$;

   $\Rightarrow \exists \tilde{\pi}' : \pi \xrightarrow{\star} \tilde{\pi}', (\tilde{\pi}, \tilde{\pi}') \in R$ and

   (a) $vis(\rho_{C}) \subseteq vis(\rho_{C})$, if $\star = pw$;
   (b) $vis(\rho_{C}) \subseteq vis(\rho_{C})$, if $\star \in \{i, s, pom\}$.

3. As item 2, but the roles of $N$ and $N'$ are reversed.

Two nets $N$ and $N'$ are $\prec$-$\tau$-bisimulation equivalent, $\star \in \{\text{interleaving, step, partial word, pomset}\}$, denoted by $N \leftrightarrow^\tau N'$, if $\exists R : N \leftrightarrow^\tau N'$, $\star \in \{i, s, pw, pom\}$.

#### ST-$\tau$-bisimulation Equivalences

**Definition 17.** An ST-$\tau$-process of a net $N$ is a pair $(\pi_E, \pi_P) s.t. \pi_E, \pi_P \in \Pi(N)$, $\pi_P \rightarrow \pi_E$ and $\forall v, w \in T_{C_E}$ $(v \prec_{C_E} w) \lor (I_{C_E}(v) = \tau) \Rightarrow v \in T_{C_P}$.

We denote a set of all ST-$\tau$-processes of a net $N$ by $ST_\tau - \Pi(N)$. $(\pi_N, \pi_N)$ is the initial ST-$\tau$-process of a net $N$. Let $(\pi_E, \pi_P), (\tilde{\pi}_E, \tilde{\pi}_P) \in ST_\tau - \Pi(N)$. We write $(\pi_E, \pi_P) \rightarrow (\tilde{\pi}_E, \tilde{\pi}_P)$, if $\pi_E \rightarrow \tilde{\pi}_E$ and $\pi_P \rightarrow \tilde{\pi}_P$. 

Definition 18. Let $N$ and $N'$ be some nets. A relation $R \subseteq ST^\tau - \Pi(N) \times ST^\tau - \Pi(N') \times B$, where $B = \{\beta \mid \beta : \text{vis}(T_C) \rightarrow \text{vis}(T_{C'})\}$, $\pi = (C, \varphi) \in \Pi(N)$, $\pi' = (C', \varphi') \in \Pi(N')$, $*$ \in \{\text{ interleaving, partial word, pomset}\}$, denoted by $R : N \leftrightarrow_{ST^\tau} N'$, $* \in \{i, pw, pom\}$, if:

1. $((\pi_N, \pi_N'), (\pi_{N'}, \pi_{N'}'), \emptyset) \in R$.
2. $((\pi_E, \pi_P), (\pi_E', \pi_P'), \beta) \in R \Rightarrow \beta : \text{vis}(\rho_C) \simeq \text{vis}(\rho_{C'})$ and $\beta(\text{vis}(T_C)) = \text{vis}(T_{C'})$.
3. $((\pi_E, \pi_P), (\pi_E', \pi_P'), \gamma : \text{vis}(\rho_C) \simeq \text{vis}(\rho_{C'}) \Rightarrow \exists \beta : \beta(\text{vis}(T_C)) = \text{vis}(T_{C'}) \in R$, and if
   - $\pi_P \xrightarrow{\pi_E, \pi_P'} \pi_E', \pi_P' \xrightarrow{\gamma} \pi_E'$, then:
     - (a) $\gamma^{-1} : \text{vis}(\rho_C) \subseteq \text{vis}(\rho_{C'})$, if $*' = \text{pw}$;
     - (b) $\gamma : \text{vis}(\rho_C) \simeq \text{vis}(\rho_{C'})$, if $*' = \text{pom}$.
4. As item 3, but the roles of $N$ and $N'$ are reversed.

Two nets $N$ and $N'$ are $*$-ST-$\tau$-bisimulation equivalent, $* \in \{\text{ interleaving, partial word, pomset}\}$, denoted by $N \leftrightarrow_{ST^\tau} N'$, if $\exists R : N \leftrightarrow_{ST^\tau} N'$, $* \in \{i, pw, pom\}$.

History Preserving $\tau$-bisimulation Equivalences

Definition 19. Let $N$ and $N'$ be some nets. A relation $R \subseteq \Pi(N) \times \Pi(N') \times B$, where $B = \{\beta \mid \beta : \text{vis}(T_C) \rightarrow \text{vis}(T_{C'})\}$, $\pi = (C, \varphi) \in \Pi(N)$, $\pi' = (C', \varphi') \in \Pi(N')$, is a pomset history preserving $\tau$-bisimulation between $N$ and $N'$, denoted by $N \leftrightarrow_{pom} ST^\tau N'$, if:

1. $((\pi_N, \pi_N'), (\pi_{N'}, \pi_{N'}'), \emptyset) \in R$.
2. $((\pi, \pi'), \beta, \gamma) \in R \Rightarrow \beta : \text{vis}(\rho_C) \simeq \text{vis}(\rho_{C'})$.
3. $((\pi, \pi'), \beta) \in R$, $\pi \rightarrow \tilde{\pi} \Rightarrow \exists \tilde{\beta} : \tilde{\beta}(\text{vis}(T_C)) = \beta(\text{vis}(T_{C'})), (\tilde{\pi}, \tilde{\pi}', \tilde{\beta}) \in R$.
4. As item 3, but the roles of $N$ and $N'$ are reversed.

Two nets $N$ and $N'$ are pomset history preserving $\tau$-bisimulation equivalent, denoted by $N \leftrightarrow_{pom} ST^\tau N'$.

History Preserving ST-$\tau$-Bisimulation Equivalences

Definition 20. Let $N$ and $N'$ be some nets. A relation $R \subseteq ST^\tau - \Pi(N) \times ST^\tau - \Pi(N') \times B$, where $B = \{\beta \mid \beta : \text{vis}(T_C) \rightarrow \text{vis}(T_{C'})\}$, $\pi = (C, \varphi) \in \Pi(N)$, $\pi' = (C', \varphi') \in \Pi(N')$, is a pomset history preserving ST-$\tau$-bisimulation between $N$ and $N'$, denoted by $N \leftrightarrow_{pom} ST^\tau N'$, if:

1. $((\pi_N, \pi_N'), (\pi_{N'}, \pi_{N'}'), \emptyset) \in R$.
2. $((\pi_E, \pi_P), (\pi_E', \pi_P'), \beta) \in R \Rightarrow \beta : \text{vis}(\rho_C) \simeq \text{vis}(\rho_{C'})$ and $\beta(\text{vis}(T_C)) = \text{vis}(T_{C'})$.
3. $((\pi_E, \pi_P), (\pi_E', \pi_P'), \gamma : \text{vis}(\rho_C) \simeq \text{vis}(\rho_{C'}) \Rightarrow \exists \beta : \beta(\text{vis}(T_C)) = \text{vis}(T_{C'}) \in R$, and if
   - $\pi_P \xrightarrow{\pi_E, \pi_P'} \pi_E', \pi_P' \xrightarrow{\gamma} \pi_E'$, then:
     - (a) $\gamma^{-1} : \text{vis}(\rho_C) \subseteq \text{vis}(\rho_{C'})$, if $*' = \text{pw}$;
     - (b) $\gamma : \text{vis}(\rho_C) \simeq \text{vis}(\rho_{C'})$, if $*' = \text{pom}$.
4. As item 3, but the roles of $N$ and $N'$ are reversed.

Two nets $N$ and $N'$ are pomset history preserving ST-$\tau$-bisimulation equivalent, denoted by $N \leftrightarrow_{pom} ST^\tau N'$.
Usual Branching $\tau$-bisimulation Equivalences For some net $N$ and $\pi, \tilde{\pi} \in \Pi(N)$ we write $\pi \Rightarrow \tilde{\pi}$ when $\exists \pi = (\tilde{C}, \tilde{\phi})$ s.t. $\pi \xrightarrow{\tau} \tilde{\pi}$ and $vis(TC') = \emptyset$.

**Definition 21.** Let $N$ and $N'$ be some nets. A relation $R \subseteq \Pi(N) \times \Pi(N')$ is an interleaving branching $\tau$-bisimulation between $N$ and $N'$, denoted by $N \leftrightarrow^\tau_\text{br} N'$, if
1. $(\pi_N, \pi_{N'}) \in R$.
2. $(\pi, \pi') \in R$, $\pi \Rightarrow \tilde{\pi} \Rightarrow$
   (a) $a = \tau$ and $(\tilde{\pi}, \tilde{\pi}') \in R$ or
   (b) $a \neq \tau$ and $\exists \tilde{\pi}'$, $\tilde{\pi}' : \pi' \Rightarrow \tilde{\pi}'$, $(\pi, \tilde{\pi}', \tilde{\pi}') \in R$, $(\tilde{\pi}, \tilde{\pi}', \tilde{\pi}') \in R$.
3. As item 2, but the roles of $N$ and $N'$ are reversed.

Two nets $N$ and $N'$ are interleaving branching $\tau$-bisimulation equivalent, denoted by $N \leftrightarrow^\tau_\text{br} N'$, if $\exists R : N \leftrightarrow^\tau_\text{br} N'$.

History Preserving Branching $\tau$-bisimulation Equivalences

**Definition 22.** Let $N$ and $N'$ be some nets. A relation $R \subseteq \Pi(N) \times \Pi(N') \times B$, where $B = \{ \beta \mid \beta : TC \rightarrow TC', \pi = (C, \phi) \in \Pi(N), \pi' = (C', \phi') \in \Pi(N') \}$, is a pomset history preserving branching $\tau$-bisimulation between $N$ and $N'$, denoted by $N \leftrightarrow^\tau_\text{phbr} N'$, if:
1. $(\pi_N, \pi_{N'}, \emptyset) \in R$.
2. $(\pi, \pi', \beta) \in R \Rightarrow \beta : vis(\rho_C) \simeq vis(\rho_{C'})$.
3. $(\pi, \pi', \beta) \in R$, $\pi \Rightarrow \tilde{\pi}$ then
   (a) $(\tilde{\pi}, \tilde{\pi}', \beta) \in R$ or
   (b) $\exists \beta, \tilde{\pi}', \tilde{\pi}' : \pi' \Rightarrow \tilde{\pi}'$, $\tilde{\beta} : vis(TC') = \beta$, $(\pi, \tilde{\pi}', \beta) \in R$, $(\tilde{\pi}, \tilde{\pi}', \tilde{\beta}) \in R$.
4. As item 3, but the roles of $N$ and $N'$ are reversed.

Two nets $N$ and $N'$ are pomset history preserving branching $\tau$-bisimulation equivalent, denoted by $N \leftrightarrow^\tau_\text{phbr} N'$, if $\exists R : N \leftrightarrow^\tau_\text{phbr} N'$.

ST-branching $\tau$-bisimulation Equivalences

**Definition 23.** Let $N$ and $N'$ be some nets. A relation $R \subseteq \text{ST}^\tau - \Pi(N) \times \text{ST}^\tau - \Pi(N') \times B$, where $B = \{ \beta \mid \beta : vis(TC) \rightarrow vis(TC'), \pi = (C, \phi) \in \Pi(N), \pi' = (C', \phi') \in \Pi(N') \}$ is an interleaving ST-branching $\tau$-bisimulation between $N$ and $N'$, denoted by $N \leftrightarrow^\tau_\text{STbr} N'$, if:
1. $((\pi_N, \pi_N), (\pi_{N'}, \pi_{N'}), \emptyset) \in R$.
2. $((\pi_E, \pi_P), (\pi_E, \pi_P), \beta) \in R \Rightarrow \beta : vis(\rho_{C_E}) \times vis(\rho_{C_E'})$ and $\beta(vis(TC_E)) = vis(TC_E')$.
3. $((\pi_E, \pi_P), (\pi_E, \pi_P), \beta) \in R$, $(\pi_E, \pi_P) \Rightarrow (\tilde{\pi}_E, \tilde{\pi}_P) \Rightarrow$
   (a) $((\tilde{\pi}_E, \tilde{\pi}_P), (\tilde{\pi}_E', \tilde{\pi}_P'), \beta) \in R$ or
   (b) $\exists \beta, (\tilde{\pi}_E, \tilde{\pi}_P), (\tilde{\pi}_E', \tilde{\pi}_P') : (\pi_E, \pi_P) \Rightarrow (\tilde{\pi}_E, \tilde{\pi}_P) \Rightarrow (\tilde{\pi}_E', \tilde{\pi}_P') \Rightarrow (\tilde{\pi}_E', \tilde{\pi}_P'), \tilde{\beta} : vis(TC_E) = \beta$, $(\tilde{\pi}_E, \tilde{\pi}_P), (\tilde{\pi}_E', \tilde{\pi}_P'), \beta) \in R$, $(\tilde{\pi}_E, \tilde{\pi}_P), (\tilde{\pi}_E', \tilde{\pi}_P'), \beta) \in R$.
4. As item 3, but the roles of $N$ and $N'$ are reversed.

Two nets $N$ and $N'$ are interleaving ST-branching $\tau$-bisimulation equivalent, denoted by $N \leftrightarrow^\tau_\text{STbr} N'$, if $\exists R : N \leftrightarrow^\tau_\text{STbr} N'$. 
History Preserving ST-branching $\tau$-bisimulation Equivalences

**Definition 24.** Let $N$ and $N'$ be some nets. A relation $\mathcal{R} \subseteq ST^\tau - \Pi(N) \times ST^\tau - \Pi(N') \times B$, where $B = \{\beta : \beta: \text{vis}(T_C) \rightarrow \text{vis}(T_{C'})\}$, $\pi = (C, \varphi) \in \Pi(N)$, $\pi' = (C', \varphi') \in \Pi(N')$ is a pomset history preserving ST-branching $\tau$-bisimulation between $N$ and $N'$, denoted by $N \equiv_{\text{pomh ST-br}}^{\tau} N'$, if:

1. $((\pi_N, \pi_N), (\pi_N, \pi_N), \emptyset) \in \mathcal{R}$.
2. $((\pi_E, \pi_P), (\pi'_E, \pi'_P), \beta) \in \mathcal{R} \Rightarrow \beta : \text{vis}(\rho_{C_E}) \equiv \text{vis}(\rho_{C'_E})$ and $\beta(\text{vis}(T_{C_E})) = \text{vis}(T_{C'_E})$.
3. $((\pi_E, \pi_P), (\pi'_E, \pi'_P), \beta) \in \mathcal{R}$, $(\pi_E, \pi_P) \rightarrow (\pi'_E, \pi'_P) \Rightarrow$
   (a) $((\pi'_E, \pi'_P), (\pi'_E, \pi'_P), \beta) \in \mathcal{R}$ or
   (b) $\exists \tilde{\beta} : (\pi'_E, \pi'_P), (\pi'_E, \pi'_P) \rightarrow (\tilde{\pi}'_E, \tilde{\pi}'_P) \Rightarrow (\tilde{\pi}'_E, \tilde{\pi}'_P), \tilde{\beta}|_{\text{vis}(T_{C_E})} = \beta, ((\pi_E, \pi_P), (\tilde{\pi}'_E, \tilde{\pi}'_P), \beta) \in \mathcal{R}$.
4. As item 3, but the roles of $N$ and $N'$ are reversed.

Two nets $N$ and $N'$ are pomset history preserving ST-branching $\tau$-bisimulation equivalent, denoted by $N \equiv_{\text{pomh ST-br}}^{\tau} N'$, if $\exists \mathcal{R} : N \equiv_{\text{pomh ST-br}}^{\tau} N'$.

### 3.3 Conflict Preserving $\tau$-equivalences

Let $\xi = (X, \prec, \#, l)$ be a LES s.t. $l : X \rightarrow Act$. We denote $\text{vis}(X) = \{x \in X \mid l(x) \in Act\}$ and $\text{vis}(\xi) = \xi|_{\text{vis}(X)}$.

**Definition 25.** Two nets $N$ and $N'$ are MES-$\tau$-conflict preserving equivalent, denoted by $N \equiv_{\text{mes}}^{\tau} N'$, if $\text{vis}(\xi(N)) = \text{vis}(\xi(N'))$.

### 4 Back-forth $\tau$-bisimulation Equivalences

In this section we propose back-forth $\tau$-bisimulation equivalences.

**Definition 26.** A sequential run of a net $N$ is a pair $(\pi, \sigma)$, where:

- a process $\pi \in \Pi(N)$ contains the information about causal dependencies of transitions which brought to this state;
- a sequence $\sigma \in T_C$ s.t. $\pi_N \xrightarrow{C} \pi$, contains the information about the order in which the transitions occur which brought to this state.

Let us denote the set of all sequential runs of a net $N$ by $\text{Runs}(N)$.

The initial sequential run of a net $N$ is a pair $(\pi_N, \varepsilon)$. Let $(\pi, \sigma)$, $(\tilde{\pi}, \tilde{\sigma}) \in \text{Runs}(N)$. We write $(\pi, \sigma) \xrightarrow{C} (\tilde{\pi}, \tilde{\sigma})$, if $\pi \xrightarrow{C} \tilde{\pi}$, $\exists \sigma \in T_C \pi \xrightarrow{C} \tilde{\pi}$ and $\tilde{\sigma} = \sigma \tilde{\sigma}$.

**Definition 27.** Let $N$ and $N'$ be some nets. A relation $\mathcal{R} \subseteq \text{Runs}(N) \times \text{Runs}(N')$ is a *-back **-forth $\tau$-bisimulation between $N$ and $N'$, **, $** \in \{\text{interleaving, step, partial word, pomset}\}$, denoted by $\mathcal{R} : N \equiv_{\text{*-back **-forth}}^{\tau} N'$, if:

1. $((\pi_N, \varepsilon), (\pi_{N'}, \varepsilon)) \in \mathcal{R}$.
2. \(((\pi, \sigma), (\pi', \sigma')) \in \mathcal{R}\)
   - (back)
     \((\pi, \sigma) \xrightarrow{\ast} (\pi, \sigma),\)
     (a) \(\text{vis}(T_{\overline{C}}) = 1, \text{ if } * = i;\)
     (b) \(\text{vis}(\overline{\mathcal{C}}) = \emptyset, \text{ if } * = s;\)
     \(\Rightarrow \exists (\pi', \sigma'): (\pi', \sigma') \xrightarrow{\ast} (\pi', \sigma'), ((\pi, \sigma), (\pi', \sigma')) \in \mathcal{R}\) and
     (a) \(\text{vis}(\rho_C) \subseteq \text{vis}(\rho_C'), \text{ if } * = pw;\)
     (b) \(\text{vis}(\rho_C) \simeq \text{vis}(\rho_C'), \text{ if } * \in \{i, s, pom\};\)
   - (forth)
     \((\pi, \sigma) \xrightarrow{\ast} (\pi, \sigma),\)
     (a) \(\text{vis}(T_{\overline{C}}) = 1, \text{ if } ** = i;\)
     (b) \(\text{vis}(\overline{\mathcal{C}}) = \emptyset, \text{ if } ** = s;\)
     \(\Rightarrow \exists (\pi', \sigma'): (\pi', \sigma') \xrightarrow{\ast} (\pi', \sigma'), ((\pi, \sigma), (\pi', \sigma')) \in \mathcal{R}\) and
     (a) \(\text{vis}(\rho_C) \subseteq \text{vis}(\rho_C'), \text{ if } ** = pw;\)
     (b) \(\text{vis}(\rho_C) \simeq \text{vis}(\rho_C'), \text{ if } ** \in \{i, s, pom\}.\)
3. As item 2, but the roles of \(N\) and \(N'\) are reversed.

Two nets \(N\) and \(N'\) are \(+\text{-back} **\)-forth \(\tau\)-bisimulation equivalent, \(+\ast\), ** \(\in\) \{interleaving, step, partial word, pomset\}, denoted by \(N \leftrightarrow^{\tau}_{\text{bis, f}} N'\), if \(\exists \mathcal{R} : N \leftrightarrow^{\tau}_{\text{bis, f}} N', * \times ** \in \{i, s, pw, pom\}.\)

5 Interrelations of the \(\tau\)-equivalences

Let us consider interrelations of all the introduced \(\tau\)-equivalences.

**Proposition 1.** Let \(* \in \{i, s, pw, pom\}.\) For nets \(N\) and \(N':\)

1. \(N \leftrightarrow^{\tau}_{\text{sub, f}} N' \iff N \leftrightarrow^{\tau}_{\text{con, b}} N';\)
2. \(N \leftrightarrow^{\tau}_{\text{bi, f}} N' \iff N \leftrightarrow^{\tau}_{\text{bi, s}} N';\)
3. \(N \leftrightarrow^{\tau}_{\text{bi, f}} N' \iff N \leftrightarrow^{\tau}_{\text{bi, s}} N' [6];\)
4. \(N \leftrightarrow^{\tau}_{\text{com, pom}} N' \iff N \leftrightarrow^{\tau}_{\text{com, pomh, f}} N' [8];\)
5. \(N \leftrightarrow^{\tau}_{\text{ST, b}} N' \Rightarrow N \leftrightarrow^{\tau}_{\text{sub, f}} N'.\)

In the following, the symbol ‘\(\_\)' will denote an empty alternative, and signs that equivalences subscribed by it are considered as that of without any subscription.

**Theorem 1.** Let \(\leftrightarrow, ** \in \{\leftrightarrow^\tau, \leftrightarrow^\tau, \simeq\}\) and \(*, ** \in \{\_i, s, pw, pom, iST, pwST, pomST, pomh, pomhST, iBR, pomhBR, iSTBR, pomhSTBR, mes, iBS f, iBSpw f, iPom f, sBS f, sBSpw f, sPom f\}.\) For nets \(N\) and \(N' : N \leftrightarrow_{\_i} N' \Rightarrow N **_{\_i} N'\) iff there exists a directed path from \(\leftrightarrow_{\_i}\) to \(**_{\_i}\) in the graph in Figure 1.

**Proof.** \((\Rightarrow)\) By Proposition 28 and the definitions of the equivalences.

\((\Leftarrow)\) An absence of additional nontrivial arrows in the graph in Figure 1 is proved by the following examples.
Fig. 1. Interrelations of the $\tau$-equivalences and their preservation by SM-refinements.
In Figure 2(a) \( N \xrightarrow{\tau}_{\text{net}} N' \), but \( N \not\xrightarrow{\tau}_{\text{net}} N' \), since only in the net \( N' \) actions \( a \) and \( b \) cannot happen concurrently.

- In Figure 2(c) \( N \xrightarrow{\tau}_{\text{ST}} N' \), but \( N \not\xrightarrow{\tau}_{\text{ST}} N' \), since for the pomset corresponding to the net \( N \) there is no even less sequential pomset in \( N' \).

- In Figure 2(b) \( N \xrightarrow{\tau}_{\text{pom,ST}} N' \), but \( N \not\xrightarrow{\tau}_{\text{pom,ST}} N' \), since only in the net \( N' \) action \( b \) can depend on action \( a \).

- In Figure 4(a) \( N \equiv_{\text{mes}} N' \), but \( N \not\xrightarrow{\tau}_{\text{mes}} N' \), since only in the net \( N' \) action \( \tau \) can happen so that in the corresponding initial state of the net \( N \) action \( a \) cannot happen.

- In Figure 3(a) \( N \xrightarrow{\tau}_{\text{pom}} N' \), but \( N \not\xrightarrow{\tau}_{\text{ST}} N' \), since only in the net \( N' \) action \( a \) can start so that no action \( b \) can begin to work until finishing \( a \).

- In Figure 3(b) \( N \xrightarrow{\tau}_{\text{pom,ST}} N' \), but \( N \not\xrightarrow{\tau}_{\text{pom,ST}} N' \), since only in the net \( N' \) after action \( a \) action \( b \) can happen so that action \( c \) must depend on \( a \).

- In Figure 4(b) \( N \xrightarrow{\tau}_{\text{pom,ST}} N' \), but \( N \not\xrightarrow{\tau}_{\text{pom,ST}} N' \), since only in the net \( N' \) action \( a \) can start so that the action \( b \) can never occur.

- In Figure 4(c) \( N \equiv_{\text{ST}} N' \), but \( N \not\xrightarrow{\tau}_{\text{ST}} N' \), since in the net \( N' \) an action \( a \) can happen so that it will be simulated by sequence of actions \( \tau a \) in \( N \). Then the state of the net \( N \) reached after \( \tau \) must be related with the initial state of a net \( N \), but in such a case the occurrence of action \( b \) from the initial state of \( N' \) cannot be imitated from the corresponding state of \( N \).

- In Figure 4(e) \( N \xrightarrow{\tau}_{\text{pom,ST}} N' \), but \( N \not\xrightarrow{\tau}_{\text{pom,ST}} N' \), since in the net \( N' \) an action \( c \) may start so that during work of the corresponding action \( c \) in the net \( N \) an action \( a \) may happen in such a way that the action \( b \) never occur.

- In Figure 4(c) \( N \equiv_{\text{mes}} N' \), but \( N \not\xrightarrow{\tau}_{\text{mes}} N' \), since only the MES corresponding to the net \( N' \) has two conflict actions \( a \).

- In Figure 4(d) \( N \equiv_{\text{mes}} N' \), but \( N \not\xrightarrow{\tau}_{\text{mes}} N' \), since unobservable transitions of the nets \( N \) and \( N' \) are labelled by different actions \( (a \) and \( b \)).

- In Figure 2(c) \( N \xrightarrow{\tau}_{\text{bsf}} N' \), but \( N \not\xrightarrow{\tau}_{\text{bsf}} N' \).

- In Figure 2(d) \( N \equiv_{\text{bsf}} N' \), but \( N \not\xrightarrow{\tau}_{\text{bsf}} N' \).

- In Figure 3(a) \( N \equiv_{\text{bsf}} N' \), but \( N \not\xrightarrow{\tau}_{\text{bsf}} N' \).

- In Figure 2(b) \( N \xrightarrow{\tau}_{\text{bsf}} N' \), but \( N \not\xrightarrow{\tau}_{\text{bsf}} N' \).

- In Figure 1, the new equivalence notions and new interrelations are printed in bold font.

6 Transition Refinement

In this section we treat the considered \( \tau \)-equivalences for preservation by transition refinements.

**Definition 28.** An SM-net is a net \( D = \langle P_D, T_D, F_D, I_D, M_D \rangle \) s.t.:

1. \( \forall t \in T_D \ | t | = | \cdot | = 1 \), i.e. each transition has exactly one input and one output place;
2. \( \exists p_{\text{in}}, p_{\text{out}} \in P_D \) s.t. \( p_{\text{in}} \not\equiv p_{\text{out}} \) and \( D = \{ p_{\text{in}} \} \), \( D^o = \{ p_{\text{out}} \} \), i.e. net \( D \) has unique input and unique output place.
Fig. 2. Examples of the $\tau$-equivalences
Fig. 3. Examples of the $\tau$-equivalences (continued)
Fig. 4. Examples of the \( \tau \)-equivalences (continued 2)
3. \( M_D = \{p_{in}\}, \) i.e. at the beginning there is unique token in \( p_{in} \).

**Definition 29.** Let \( N = \langle P_N, T_N, F_N, l_N, M_N \rangle \) be some net, \( a \in l_N(T_N) \) and \( D = \langle P_D, T_D, F_D, l_D, M_D \rangle \) be SM-net. An SM-refinement, denoted by \( \text{ref}(N, a, D) \), is a net \( \overline{N} = \langle \overline{P}_N, \overline{T}_N, \overline{F}_N, \overline{l}_N, \overline{M}_N \rangle \), where:

- \( \overline{P}_N = P_N \cup \{\langle p, u \rangle \mid p \in P_D \setminus \{p_{in}, p_{out}\}, u \in l_N^{-1}(a)\} \);
- \( \overline{T}_N = (T_N \setminus l_N^{-1}(a)) \cup \{\langle t, u \rangle \mid t \in T_D, u \in l_N^{-1}(a)\} \);
- \( \overline{F}_N(\overline{x}, \overline{y}) = \begin{cases} F_N(\overline{x}, \overline{y}), & \overline{x} = (x, u), \overline{y} = (y, u), u \in l_N^{-1}(a); \\ F_N(\overline{x}, \overline{u}), & \overline{y} = (y, u), \overline{x} \in \bullet u, u \in l_N^{-1}(a), y \in p^{\bullet}; \\ F_N(u, \overline{y}), & \overline{x} = (x, u), \overline{y} \in \bullet u, u \in l_N^{-1}(a), x \in p^{\bullet_{out}}; \\ 0, & \text{otherwise}; \end{cases} \)
- \( \overline{l}_N(\overline{u}) = \begin{cases} l_N(\overline{u}), & \overline{u} \in T_N \setminus l_N^{-1}(a); \\ l_D(t), & \overline{u} = (t, u), t \in T_D, u \in l_N^{-1}(a); \end{cases} \)
- \( \overline{M}_N(p) = \begin{cases} M_N(p), & p \in P_N; \\ 0, & \text{otherwise}. \end{cases} \)

An equivalence is preserved by refinements, if equivalent nets remain equivalent after applying any refinement operator to them.

**Theorem 2.** Let \( \leftrightarrow \in \{\equiv^\tau, \equiv^\tau_{ref}, \simeq\} \) and \( * \in \{\_\_ i, s, p, p_{w}, p_{w} s, p_{w} s T, p_{w} s T_{w}, p_{w} s T_{w} b_{r}, p_{w} s T_{b} b_{r}, p_{w} h_{b} r_{e}, p_{w} h_{b} r_{e} s, i b_{s} f, i b_{w} p_{f}, i b_{p} m_{f}, s b s f, s b p_{w} s, s b p_{w} s m_{f}\} \). For nets \( N, N' \) s.t. \( a \in l_N(T_N) \cap l_N(T_{N'}) \) and SM-net \( D : N \leftrightarrow_*, N' \Rightarrow \text{ref}(N, a, D) \leftrightarrow_* \text{ref}(N', a, D) \) iff the equivalence \( \leftrightarrow_* \) is in oval in Figure 1.

7 Conclusion

In this paper, we supplemented by new ones and examined a group of basic \( \tau \)-equivalences and back-forth \( \tau \)-bisimulation equivalences. We also compared them on the whole class of Petri nets as well as on subclass of sequential nets. All the considered \( \tau \)-equivalences were checked for preservation by SM-refinements.

Further research may consist in the investigation of \( \tau \)-variants of place bisimulation equivalences [2] which are used for effective semantically correct reduction of nets. In [1] a notion of interleaving place \( \tau \)-bisimulation equivalence was proposed, and its usefulness for simplification of systems was demonstrated.

**References**

Local Stores in C++

Mark Utting and James Dunwoody

Department of Computer Science
The University of Waikato
Private Bag 3105, Hamilton, New Zealand
{marku, jld3}@cs.waikato.ac.nz

Abstract. Local stores are a good technique for managing aliasing of pointers. They make reasoning easier and act as useful documentation in programs and specifications. This paper describes several interesting design issues that arise when local stores are incorporated into an object-oriented language and gives an idea of how usable the resulting language is. The same techniques and design alternatives should be applicable to other object-oriented languages, such as Java and object-oriented refinement calculi.

1 Introduction

Recently, there has been an increasing amount of research on axiomatic semantics and proof theories for object-oriented languages. For example, work that we are aware of includes:

- the Modula-3 and Java Extended Static Checking projects at DEC SRC [11] (both languages are translated into an object-oriented extension of Dijkstra's guarded commands [6]).
- the Java Modelling Language (FML), which adds pre/post style specification constructs to Java [9]
- a Hoare logic for a Java subset [18]
- a Hoare logic for a simple object-oriented language [2]
- an axiomatic semantics of the POOL language, which includes pointers to objects [5, 1].
- a weakest precondition semantics for Oberon [3]
- a weakest precondition semantics for an object-oriented refinement calculus [23, 22].

Most of these (all except the last two) use a single global store for all objects. This use of a single store complicates reasoning, because any two object pointers are potentially aliased. The possibility of aliasing typically generates many if-then-else cases in weakest preconditions, and results in an exponential explosion in the size of formulae if it is not controlled in some way.

There are several techniques for reducing the overhead of reasoning about aliasing within a global store:
the type system of the language may ensure that certain types of pointers cannot be aliased. For example, in a single inheritance language like Oberon, $a : A$ and $b : B$ cannot be aliased unless $A$ inherits from $B$ or $B$ inherits from $A$. However, this property does not hold in most languages with multiple inheritance, because it is always possible to create a new subclass that inherits from both $A$ and $B$. Similarly, in Java, if $A$ or $B$ are interfaces, then the type system does not prevent $a$ and $b$ being aliased, because multiple inheritance of interfaces is allowed.

- let definitions can be used to reduce the size of expressions, by factoring out each if-then-else subexpression. However, this does not reduce the actual number of cases that must be considered during reasoning. It may also decrease readability of expressions.

- in most OOLs, data fields of objects have unique names. This means that an assignment to $x.a$ cannot affect the value of $y.b$ (even if $x = y$), because $a$ and $b$ must be distinct data fields, because the names $a$ and $b$ are syntactically different. This technique is very effective at reducing aliasing cases within expressions that reference data fields (a common scenario). However, this technique is not sound for languages like Eiffel [12] and Sather [15, 20] which allow subclasses to rename data fields. Nor is it sound for languages that do not distinguish between parameterless function calls and data field reads¹ (for example, in Eiffel, $y.b$ could be a call to a function $b$ whose result depends upon the value of the data field $y.a$, which might be aliased with $x.a$). It does appear to be sound in C++ and Java, where function calls must be written as $y.b()$, so are syntactically distinguished from data field accesses.

These techniques seem to be enough to limit alias reasoning to a manageable level in the Extended Static Checking prover [11]. However, we believe that local stores are worth exploring, because they will further simplify reasoning about aliasing, they should be applicable to a wider range of object-oriented languages than the above techniques, and they are useful within specifications for expressing frames.²

This paper describes the basic ideas behind local stores, and presents several design alternatives that arise when local stores are incorporated into an object-oriented language. A key issue for the viability of local stores is how usable they are for programmers, so we address this in Section 4 by comparing several typical programs with and without local stores.

We chose to integrate local stores into C++ because it allows user-defined types to use the built-in expression syntax, it is a worst-case for formal methods, it is widely known and used, and its macro language can be used to prototype new syntactic constructs that might require compiler support in other languages.

¹ Meyer points out that languages with good information hiding should not distinguish between these, so that implementors of classes have maximum flexibility in their choice of data representations. [12]

² The frame of a method specification indicates which objects may be modified by the method.
2 Local Stores: Background

The basic idea of local stores is that programmers should explicitly group objects into many small local stores, rather than one large (implicit) global store. This basic idea was first developed fully in the *collections* of Euclid [8]. Their rationale is worth repeating:

“Thus, the programmer can partition his dynamic variables and pointers into separate collections to indicate some of his knowledge about how they will be used; the verifier is assured that pointers in different collections can never point to overlapping variables.” [17, Page 14]

Three of the main advantages of using local stores are:

1. simplified reasoning about aliasing, because pointers into distinct local stores cannot be aliased.
2. local stores are an additional program annotation, and can aid program clarity. This is because, with local stores, the program source clearly indicates application-specific clustering of related objects, which is useful information for readers. Furthermore, these clusters of objects (the local stores) are often natural candidates to use in the frames of method specifications.
3. a higher degree of compiler optimization is possible, because pointers into distinct local stores cannot interfere. Current compilers must try to infer these non-aliasing properties automatically [10, 4], which is a difficult task and much aliasing information is lost. The addition of explicit local stores to a language should make such inference simpler and more exact, thus allowing better optimization.

Previous work by Utting on local stores [24] extended the collections of Euclid in two basic ways:

- It added a *transfer* operation which allow objects to be moved from one local store to another (in Euclid, each pointer variable was associated with a fixed local store at the point of declaration). This allows the partitioning of the global store to be dynamic, rather than static.
- It noted that collections (local stores) do not have to be designed into a language, but can be added retrospectively as a user-defined abstract data type that provides the necessary dereference and update operations.

A formal specification of an appropriate ADT for local stores is shown in Fig. 1. The language used for describing programs and ADTs in this paper is the refinement calculus notation [13], extended with a simple ADT construct:

\[
\text{ADT } T \triangleq \text{Type Init Operations}.
\]

*T* is the name of the ADT, *Type* is the state space of the ADT instances, *Init* is a predicate that specifies the initial state of each instance and *Operations* is a set of named procedures whose formal parameters may be instances of type *T*. Within expressions and formulae, we use *Z* notation [21].
\[\text{Val} \quad \text{(the set of storable values)}\]
\[\text{Ptr} : \mathbb{P} \text{Val} \quad \text{(the set of pointers is a subset of Val)}\]
\[\text{nil} : \text{Ptr}\]

**ADT** \(\text{Store} \equiv \text{Ptr} \setminus \{\text{nil}\} \Rightarrow \text{Val}\)

**initially** \(s : \text{Store} \equiv s = \emptyset\)

**procedure** \(\text{add}(\text{ref} \ s : \text{Store}, \ \text{value} \ v : \text{Val}, \ \text{ref} \ p : \text{Ptr})\)
\[\hat{s}, p : \begin{cases} \text{true}, & p \notin \text{dom} \ s_0 \\ \text{false}, & p \in \text{dom} \ s_0 \end{cases}\]

**procedure** \(\text{update}(\text{ref} \ s : \text{Store}, \ \text{value} \ p : \text{Ptr}, \ \text{value} \ v : \text{VAL})\)
\[\hat{s} : \begin{cases} \text{true}, & p \notin \text{dom} \ s, s = s_0 \cup \{p \mapsto v\} \\ \text{false}, & p \in \text{dom} \ s \end{cases}\]

**procedure** \(\text{deref}(\text{value} \ s : \text{Store}, \ \text{value} \ p : \text{Ptr}, \ \text{ref} \ v : \text{Val})\)
\[\hat{v} : \begin{cases} \text{true}, & p \in \text{dom} \ s, v = s(p) \\ \text{false}, & p \notin \text{dom} \ s \end{cases}\]

**procedure** \(\text{transfer}(\text{ref} \ s, t : \text{Store}, \ \text{value} \ \text{ptrs} : \mathbb{P} \text{Ptr})\)
\[\hat{s}, t : \begin{cases} \text{true}, & \text{ptrs} \subseteq \text{dom} \ s, t = s_0 \cup \{p \mapsto v\} \\ \text{false}, & \text{ptrs} \cap \text{dom} \ s_0 = \emptyset \end{cases}\]

**Fig. 1.** The Local Store ADT

Note that a local store \(s\) is modelled as a function from pointers to values, so to dereference a pointer \(p\) within specification expressions, we can simply use function application, writing \(sp\). Of course, programs within the executable subset of the refinement calculus language are restricted to invoking operations of the \(\text{Store}\) ADT, so must use the \(\text{deref}\) procedure there.\(^3\)

The collections of Euclid were monomorphically typed. In object-oriented languages, we need to relax this to at least allow subtype and supertype objects to be stored in the same collection. For maximum generality, the ADT in Fig. 1 uses untyped pointers and objects, but when applying this to C++ (Section 3), we have found it convenient to use a template class, to reduce the number of typecasts that would be needed otherwise and to take advantage of C++ type checking.

### 2.1 Global Pointers and Efficient Implementations

Careful inspection of the \(\text{transfer}\) operation shows that it is technically infeasible, because it must ensure that \(\text{ptrs}\) and \(\text{dom} \ t_0\) are disjoint, but it cannot modify either of those sets. It is specified this way to ensure that all pointers allocated by the local stores must be globally unique. In other words, although the \(\text{transfer}\) operation is infeasible in isolation, the ADT as a whole is feasible.

\(^3\) Function notation would make this more convenient, but we shall not bother with it for this paper. In Section 3, we will discuss appropriate syntax for the \(\text{deref}\) procedure in C++.
It is also important to note that our local stores do not provide any operations for testing whether or not a pointer $p$ belongs to a particular local store $s$. Instead, a program must know that $p \in \text{dom } s$ before it can legally dereference $p$ (the precondition of $\text{deref}$ requires this). The lack of these runtime testing operations means that we can data refine all the local stores in a program into a single global store that does not need to record which pointers belong to which local stores. This result is proved in [24]. It means that we can implement all pointer operations efficiently in the standard global-store fashion.

This clearly shows that local stores are primarily a notation for documenting the usage of pointers and limiting possible aliasing patterns. The local store notation can be removed, leaving a standard program with identical aliasing patterns, but those aliasing patterns are less clearly documented because they all occur within a single global store.

### 3 Designing Local Stores into C++

The first issue that arises when incorporating local stores into C++ is that the $\text{update}$ operation in Fig. 1 is not the usual style of updating objects in C++. It takes the updated copy of the object as a call-by-value parameter and copies it into the store, overriding the old value. However, in C++, and most other object-oriented languages, objects are typically modified in situ, so that the identity of the object is not changed. That is, if $p$ is a pointer to some object that has a method $m()$, in C++ we usually write $p->m()$ to apply method $m()$ and update the object.

For this reason, we changed the interface of the $\text{update}$ operation so that its third parameter is a method call (name plus arguments, $m(e_1 \ldots e_n)$), rather than an updated copy of the object. Within the $\text{update}$ operation, the given method is applied to the appropriate object in the store, to mutate the contents of that object. We also renamed $\text{update}$ to $\text{mutate}$ to reflect this change.

A similar issue arises with $\text{deref}$. In C++, when we dereference an object, we usually want to access one of its data fields, rather than return a copy of the object. So, we changed $\text{deref}$ so that it takes an extra parameter which is the name of the data field we want to read, and it returns the value of that field.

Of course, in C++, these new parameters (a method call and a data field name) are not valid syntactic objects in isolation, so cannot legally be passed as parameters. So we defined C macros for the $\text{mutate}$ and $\text{deref}$ methods ($\text{M}$ is a method call and $\text{F}$ is a data field name).

```c
#define stmutate(p,M) (mutate(p)->M)
#define stderef(p,F) (deref(p)->F)
```

Given a $\text{Store}$ object called $s$, these macros allow us to write

- $s.stmutate(p,m(1,2))$ for $s.mutate(p)->m(1,2)$
- $s.stderef(p,f)$ for $s.deref(p)->f$
Note that the names `std::mutate` and `std::deref` must be reserved for this purpose throughout the entire program, because the macros are not type-sensitive, so will be applied everywhere. In fact, we also use these macros to get better error messages when store errors are detected, by passing the source code file name and line number into the `mutate` and `deref` methods.

```c
#define std::mutate(p, M) (mutateWithErrMsg(p, __LINE__, __FILE__) -> M)
#define std::deref(p, F) (derefWithErrMsg(p, __LINE__, __FILE__) -> F)
```

This is handy, but would be unnecessary if C++ provided a standard way of displaying a stack traceback when an error occurs.

The main attraction of this macro-based `mutate/deref` approach is that the effect of the `mutate` method is clearly limited to applying one method call and the effect of the `deref` method is limited to reading one data field. The underlying `S::mutate(p)` and `S::deref(p)` methods both return a C++ pointer to an object within the store s, but `deref(p)` returns a `const` pointer, while `mutate(p)` does not. Distinguishing between mutates and reads in this way seems attractive, and reduces the temptation for the programmer to circumvent the store protection by saving the C++ pointer returned by `mutate(p)` and using it to perform several mutations. However, it has the disadvantage of requiring macros – 1970's technology.

An alternative that we have explored is to drop the distinction between mutation and reading of an object and just provide one general `deref(p)` method. It then becomes possible to overload the standard C array operator for this dereference operation. That is, we write `s[deref(p)]` as `s[p]`. We define the overloaded array operator to return a C++ `reference` to an object within the store\(^4\) and write `s[p].f` to read a data field `f` and `s[p].m(1,2)` to call a method `m` with arguments. The ability to use familiar array syntax is a strong recommendation for this approach. It is also more concise than the `mutate/deref` methods, which is an important consideration for something that will be so frequently used.

Of the two alternatives, we prefer the readability and familiarity of the array syntax, even though it looses the clear syntactic distinction between mutating and reading objects and delimits the scope of mutations less clearly.

Fig. 2 shows the declaration of the C++ `Store` class, using the array syntax (the `mutate/delete` syntax is also shown, but is commented out). The string passed to the constructor is just a name for the store, so that error messages can give some indication of which store has raised an error. Comparing this `Store` class with the specification in Fig. 1, you will see that we have also added a `remove` operation that does a C++ `delete` on an object (because C++ does not usually have automatic garbage collection) and split the `transfer` operation into two common cases (transfer a single object and transfer all objects). This avoids passing a set of pointers to `transfer`, and seems to be sufficiently flexible in practice.

\(^4\) We could also have used references rather than pointers in the `mutate/deref` approach, but the macro syntax actually hides the distinction, so it does not really matter which we choose.
template<class ValPtr, class Val>
class Store{
public:
    Store(char *);
    Store();
    ValPtr add(Val *v);
    Val &operator[](ValPtr ref);
    // const Val *deref(ValPtr v);
    // Val *mutate(ValPtr v);
    void remove(ValPtr v);
    void transfer(ValPtr, Store<ValPtr, Val> &);
    void transferAll(Store<ValPtr, Val> &);
};

Fig. 2. Declaration of C++ Store class

Note that Store is a template class, with each store mapping ValPtr values to objects of type Val. We make it a template class in order to allow the C++ compiler to do more precise typechecking. That is, rather than having a single Ptr type for all pointers into all stores, each different type of instantiation defines a new type of pointer. We use the following macros to hide the details:

#define STORETYPE(T) class T##OBJ {}
typedef T* T##Ptr
#define STORE(T) Store<T##Ptr,T>

For example, we declare and use several stores containing ListNode objects as follows:

STORETYPE(ListNode); // done once;
STORE(ListNode) s1("Mark’s list");
STORE(ListNode) s2("James’ list");

// put two nodes into Mark’s list, with p being the head.
ListNodePtr p = s1.add(new ListNode(42));
s1[p].next = s1.add(new ListNode(64));

// put a circular structure into James’ list.
ListNodePtr q = s2.add(new ListNode(53));
s2[q].next = q; // create a loop.

This defines a type called ListNodePtr, which abbreviates (ListNodeOBJ *), where ListNodeOBJ is an empty class created just for the typechecker’s benefit. The two stores s1 and s2 both map ListNodePtr values to ListNode objects. Both the assignments shown are legal. However, it would be illegal to dereference p using s2, because the precondition of the deref operation requires p ∈ dom s2.
The assignment \( s1[p].next = q \) would also be legal, even though it would result in a data field of an object owned by \( s1 \) (i.e., list node number 42) pointing into a different store \( s2 \). This just means that we would have to use store \( s2 \) to dereference that data field, rather than store \( s1 \).

What does it mean to say that a dereference (or mutate) operation is illegal? The obvious answer is just that the precondition is false, so the results of such a call are undefined. In other words, it is a programmer’s responsibility, or a program verifier’s responsibility, to make sure that such illegal calls never occur. However, we have also developed checking implementations of the Store class that explicitly check all such preconditions at runtime and print meaningful error messages whenever one is violated. This is useful when developing or debugging a program.

Our current checking implementation catches the following errors:

- attempting to dereference a NULL pointer;
- attempting to dereference a pointer that is not in the domain of that store;
- dangling references (dereferencing a pointer to an object that has been deleted);
- memory leaks (the destructor of the store raises an error if the store is not empty). This check would not be desirable in a language with automatic garbage collection, but catches quite a few errors in C++ programs.

All operations of the checking implementation are done in constant time\(^5\). However, the checking implementation obviously has significant overheads, because pointer dereferencing is usually extremely efficient. So we provide a release implementation that defines all store operations to be the standard pointer operations, inlined, and this removes all overheads (and all checking!).

The last issue to discuss is: how should we check that local stores really do partition the objects in the system? Recall that the key goal of local stores is to make it syntactically obvious whether or not aliasing can occur, and this requires that each local store manage a separate set of objects. If an object could be inserted into two stores:

\begin{verbatim}
STORE(ListNode) s1("Mark’s list");
STORE(ListNode) s2("James’ list");

// create a C++ pointer to a list node.
ListNode *p = new ListNode(42);

ListNodePtr q = s1.add(p);
ListNodePtr r = s2.add(p);
\end{verbatim}

\(^5\) To allow the transferAll to be done in constant time, we use a global hashtable that maps each pointer to an index into a table of store identity numbers. To transfer all pointers from store \( s1 \) into store \( s2 \), we simply overwrite \( s1 \)'s entry into that table with the value of \( s2 \)'s entry.
then this property would be destroyed, because mutating the $q$ object in store $s1$ would have a side-effect on the $r$ object in store $s2$.

We have two strategies for ensuring that such situations cannot arise.

1. Use a single global hashtable to record which pointers belong to which stores. This means that an attempt to add a pointer $p$ twice can easily be detected (and in constant time).

2. Redefine the `add` method so that it invokes the C++ `new` operator. To do this, we rename `add` to something obscure like `dont_call_this_add` and define a `stnew` macro that must be used to add objects into stores.

   ```
   #define stnew(C) dont_call_this_add(new C)
   ```

   This means that only freshly created objects can be added into stores. This would be our preferred strategy if we could incorporate it into a language without having to rely on macros.

4 Examples

This section gives some example code that uses stores, to demonstrate their usability. The example in Fig. 3 is a simple linked list class, given both in standard C++, and using local stores (with the array syntax and the `stnew` version of `add`).

Note that `p1.addAll(p2)` removes all projects from the `p2` project list and adds them into `p1`. It does this simply by linking the two linked lists together. However, in the local store version, we also want all the reachable `node` objects to belong to `p1`, so we call `transferAll` to move all `p2`'s `node` objects out of `p2`'s local store and into `p1`'s local store. With the checking implementation this `transferAll` operation takes constant time, with the release version, it takes zero time.

The other interesting difference is that some methods must take an extra store argument, so that pointers (like `p2`) can be dereferenced. This is inconvenient, and could perhaps be eliminated in many cases if we developed rules for passing objects like `p2` by reference, after ensuring that they were not aliased with other parameters. This is a topic for future research – the focus of this paper is pointer aliasing.

Our second example (Fig. 4) is a quicksort program, that sorts an array of pointers to `Person` objects, but must dereference those pointers to compare objects. The natural way of using stores in this program is to use a single local store for all the objects in the array. However, to push local stores to the limit, we take a more extreme approach here, and use separate local stores for the left and right partitions of the array, at each level of recursion. Upon entering each recursive level, stores are created to manage the sub-arrays. Upon exiting the recursion, the objects in these stores are transferred back into the parent store.
5 Preventing Aliasing of Local Stores

The approach we take to controlling aliasing is essentially an application of the *syntactic control of interference* approach of Reynolds [19]. Aliasing is a kind of interference (interaction) between assignments and expressions. The key principle of Reynolds approach is that all such interference must go through a common identifier, so that the interference is syntactically obvious. That is, an assignment $L := R$ can only modify the value of an expression $E$ when $L$ and $E$ mention a common identifier, typically a common local store in our case. It is
int split(PersonPtr A[], int l, int r,
STORE(Person) &lStore,
STORE(Person) &rStore,
STORE(Person) &store) {
int m = rand() % (r-l) + 1;
// swap A[l] and A[m]
PersonPtr temp = A[l];
A[l] = A[m];
A[m] = temp;
for (int j = l + 1; j <= r; j++) {
    if (*store[A[j]] < *store[A[l]]) {
        i++;
        store.swap(A[j], lStore);
        // swap A[l] and A[j]
        temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    } else {
        store.swap(A[j], rStore);
    }
}
// swap A[l] and A[i]
int i = split(A, l, i-1, lStore);
qsrt(A, l, i-1, lStore);
qsrt(A, i+1, r, rStore);
lStore.swapAll(store);
rStore.swapAll(store);
}

void qsort(PersonPtr A[], int l, int r, STORE(Person) &store) {
if (r-l >= 1) {
    STORE(Person) lStore("Left store");
    STORE(Person) rStore("Right store");
    int i = split(A, l, r, lStore, rStore, store);
    qsort(A, l, i-1, lStore);
    qsort(A, i+1, r, rStore);
    lStore.swapAll(store);
    rStore.swapAll(store);
}
}

void quicksort(PersonPtr A[], STORE(Person) &store, int n) {
    qsort(A, 0, n, store);
}

Fig. 4. Quicksort procedures using local stores
this simple syntactic check that allows the majority of aliasing if-then-else cases to be resolved immediately, and gives efficient reasoning.

For this property to hold, we must ensure that local stores with different names can never be aliased. Euclid imposed quite strict restrictions (sometimes requiring runtime checks) on the parameters of procedure calls to ensure that no formal parameters could be aliased. Euclid also ensured that the global variables accessed within a procedure are not aliased with any parameters. Developing similar rules for a class-based object-oriented language is tricky, because methods refer to data fields via an implicit this variable, but the system may also contain other pointers to the same object, so aliasing of those data fields needs to be addressed.

To illustrate how things can go wrong if two stores are aliased, or if the this object is aliased with an object in a store, assume that in Fig. 3 we have a store $s$ that contains a project object pointed to by $p$. Then the method call $s[p].addall(p,s)$ does not have the expected effect, because within the addall method, this and spl[p2] are aliased.

In the remainder of this section, we shall sketch a set of rules for ensuring that two stores can never be aliased in our C++ system. To simplify the presentation, without loss of generality, we make lots of simplifying assumptions:

1. Local stores can be created within the data fields of objects and as local variables, but they cannot be copied.
2. We partition the set of all C++ values into three groups: atomic immutable values, stores, and other (non-store) objects. To make the distinction clear, we shall reserve variable names that start with $s$ for store variables.
3. We require all non-store objects to be allocated on the heap.
4. Global variables are not allowed (they can be simulated by passing extra pointer and store parameters).
5. All arguments are passed using call-by-value, except for stores, which are always passed using call-by-reference.
6. All pointer operations are done via the store methods presented above and the references returned by $s[p]$ are not stored in variables etc.
7. All method calls are factored out of expressions into separate statements.

With these simplifications, we can model the expressions $e$ of C++ with the following simple grammar. Note that $e$, $e_1$, $e_2$ etc. are expressions, $se$ is any expression that returns a store object, $f$ is any data field name, $v$ is any (non-store) variable, $s$ is any variable of type Store, $c$ is any constant (a literal immutable value, like 3) and $op$ is any of the builtin C++ operators (not a method call).

$$ e :: = \text{se}[c] \mid e.f \mid v \mid s \mid e \mid e_1 op e_2 $$

Consider a typical method call, with $n$ call-by-value parameters, and $m$ store arguments passed by reference.

$$ se_0[e_0], m(e_1, \ldots, e_n, se_1, \ldots, se_m). $$
The actual method that this will invoke will depend upon the dynamic class of $se_0[e_0]$, but its formal parameters will have the form $m(v_1, \ldots, v_n, s_1, \ldots, s_m)$. Of course, $e_0$ is an implicit parameter to $m$, corresponding to the formal parameter $this$. Similarly, we shall model the $se_0$ store as an implicit parameter that is passed to $m$, with the corresponding formal parameter being called $sthis$ (for ‘store of this’). Within method $m$, we assume that all accesses of data fields and methods of $this$ are expanded out so that $f$ is written as $sthis[\textit{this}].f$ (for the moment, $\textit{this}$ is the only allowable index into the $sthis$ store).

Our goal is to ensure that within $m$, none of the stores $s_1 \ldots s_m$ are aliased and none of the objects accessible from them are aliased with the $\textit{this}$ object. Note that it is fine for several $e_1 \ldots e_n$ expressions to be identical, because they are passed using call-by-value.

To ensure this, it is sufficient for expressions $se_0$ to $se_m$ to satisfy:

$$(\forall i, j : 0 \ldots m \cdot i \neq j \Rightarrow se_i \neq se_j)$$

where the non-interference relation $\#$ is the smallest relation that satisfies the following inference rules:

$$\begin{align*}
&c \# e \\
&\frac{}{e_1 \# e_2} \\
&\frac{s_1, s_2 \text{ distinct}}{s_1 \# s_2} \\
&\frac{e_1 \# e_2}{se \# e_2} \\
&\frac{se[e] \# e_2}{e_1 \# e_3} \\
\end{align*}$$

These non-aliasing calling restrictions are quite strict. As in Euclid, they ensure that formal parameters are never aliased. They also forbid method calls like $se_0[p].m(se_1, se_1[p], s2)$, where $s2$ is a data field that contains a local store, because within $m$ the second formal parameter would be aliased with a subcomponent of the first parameter.

A limitation of the above rules is that the $sthis$ store can only be used to access the $\textit{this}$ pointer. This means that the method cannot access other sibling objects in the $sthis$ store. This restriction was made because C++ does not actually pass the $sthis$ store implicitly, so we must pass it explicitly if we wish to use it within the method. (But when the $\textit{this}$ object is the only $sthis$ object that the method accesses, passing $sthis$ explicitly is a pain, so we made this special case).

We can remove this restriction by following the convention that it is allowable for one of the explicit store parameters ($se_i$) to be identical to the store of the $\textit{this}$ object ($se_0$), provided that the corresponding formal parameter $s_i$ is called $sthis$. This preserves the key property of all formal store parameters $s_1 \ldots s_m$ being bound to distinct stores, but allows full access to $sthis$. Note that if none of the method’s formal store parameters are called $sthis$, then the only legal use
of \textit{this} is to dereference \textit{this}, as before (and \textit{seq} must be distinct from all other
store parameters as before).

Ideally, these non-aliasing call restrictions would be checked by the compiler,
plus either runtime checks or theorem proving for any \( e_1 \neq e_2 \) tests that are
not decidable at compile time. However, for this paper we simply verified the
restrictions by hand.

6 Conclusions

We have shown that it is practical to incorporate local stores into an object-
oriented language. The syntactic overhead is not huge – no more than the array
syntax that programmers are already familiar with. However, extra store param-
eters must sometimes be passed to procedures, which is a little inconvenient.

In the programs that we have written using local stores, it seems reasonably
easy to find reasonable partitions of the global store into several local stores.
Typically, each data structure has its own local store. In our quicksort example,
we took a more extreme approach and introduced extra stores at each level of
recursion, but we think that this level of partitioning is unnecessarily fine and
would give little extra advantage when reasoning about the program.

Where an object \( x \) uses several other objects within its implementation, but
they are not visible in its interface, those \textit{rep} objects can be put into their own
local store which is owned by \( x \). This is perhaps the most important use of local
stores in practice because it makes it clear that updates of those \textit{rep} objects are
internal to \( x \) and do not affect the global store or other objects in the system.
Indeed, this is the case that is the focus of most of the related research: Islands [7],
Demesnes [25] and Flexible Alias Protection [14, 16].

Once programs have been written to use local stores, it is possible to provide
an implementation that does runtime checking of all pointer operations and en-
sures that the local stores do partition the objects in the system. Our experience
so far suggests that these runtime checks do catch a lot of errors, but most of
them would be caught by any safe language – very few of the errors seem to be
partitioning problems such as accessing an object via the wrong store. We are
not sure if this is good or bad :) .

Our intention is not for C++ programmers to adopt this local store technique.
We just wanted to prototype its feasibility. The next steps are to extend the
techniques to handle special cases more conveniently, then to build local stores
directly into a language with compiler support for the non-alias checks and
evaluate its usability on larger case studies.

Another area of future work is incorporating the theory with the Java \textit{flexible
alias protection} [14] system, which is essentially a restricted version of local
stores that is statically checkable. This also has the advantage that the extra
store parameters referred to above are only needed at compile time.
References


Teaching Formal Methods Lite

Mark Utting and Steve Reeves
Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand
{marku,stever}@cs.waikato.ac.nz

Abstract. We describe a new approach to teaching formal methods, based on the philosophy that “Formal Methods are Useful”. As expected, this philosophy has led us to focus on formal specification, rather than code verification. However, it also led to a strongly tools-based approach, with practical work that reinforces traditional software engineering skills, such as unit and system testing, inspection and defensive programming with assertions. Our two main results are to identify several practical uses of formal specifications that are not widely practiced or taught, and to demonstrate that teaching them results in a more interesting and relevant formal methods course.

1 Introduction

Like many other universities, The University of Waikato offers a course on formal methods as part of the computer science curriculum. The course we teach is entitled “Advanced Software Engineering”, is a compulsory subject in the software engineering stream of the 4 year BCMS degree and an optional subject in other streams. It generally attracts about 20 to 25 students. Topics covered in the course usually include formal specification using $Z$ [17], code verification using Floyd-Hoare logics [7] and simple forms of proof such as calculating preconditions, verifying invariants and proving simple safety properties.

Student reactions to the course were mixed, with a few being enthusiastic about the approach, but many saying that they see it as “irrelevant” and “not practical”. Course evaluations in 1997 caused us concern, as the overall student rating of the course was 2.9 on a scale of 1 to 5 (1=Excellent, 3=Satisfactory, 5=Poor), with 64% percent of students stating that they would not encourage others to take the course, and 41% stating that their interest in the subject was reduced or reduced greatly by taking the course (23% unchanged, 36% increased). Of equal concern was that, when asked “How valuable do you think this course has been for you?”, only 18% of students replied extremely or very valuable, 41% said moderately valuable and 41% said slightly valuable or not valuable.

Consequently, we decided to redesign the course to focus on practical applications of formal methods, as described in Sections 2 through 4. We ran the redesigned course in the first semester of 1998 with positive outcomes, but also
some surprising findings, and that experience is described in Section 5. Our conclusions and areas for further improvement are given in Section 6.

2 Course Design Philosophy

Although typical software engineers might see formal methods as being useful for safety-critical systems, where a high degree of quality assurance is required, they usually do not believe that it is cost-effective to apply formal methods to more typical software applications. In many ways, we agree with them. We believe that formal development or verification of code is currently too time-consuming to be cost-effective, and we believe that many possible benefits of formal specification are currently not being practiced or taught.

The most commonly cited benefits of formal specification are:

- the process of writing the formal specification leads to a deeper understanding of the system being specified and typically uncovers many sources of ambiguity and incompleteness in the informal requirements.
- the formal specification forms a good basis for detailed design and code development (either formal or informal), because it is a concise precise description of the system that is to be built.

While we personally agree that these are major benefits, it is not easy to present convincing objective evidence for the cost-effectiveness of formal specification from these alone. The improved understanding and precision that result from formal specification is difficult to measure quantitatively, so claims about its benefits are often subjective. The same is true for the second benefit, given that our typical software engineers will be developing code informally, rather than formally.

Another criticism is that it is almost as easy to introduce mistakes into a formal specification as it is into a program, so there is no guarantee that a formal specification will be a more accurate reflection of a client’s real requirements than a program would be. Current formal specification practice is not strong on validation techniques, other than type checking to detect obvious mistakes and perhaps inspection to cross-check the formal specification against informal requirements documents.

Our approach to answering these questions is to emphasise other potential benefits of formal specifications, that are not widely practiced or taught. Current practice is to write a formal specification, type check it, then move on to later (informal) stages of development, with the formal specification treated as a reference document only. In contrast, our dream is to see the formal specification become the central document in the whole software development environment, being continually used by various tools throughout the software lifecycle. Even though code development is done informally, there are useful ways of relating specification and code. In other words, our philosophy is that, having gone to the effort of writing a formal specification, we should then get as much benefit out of it as possible.
Here are some of the specific techniques that we believe should be practiced more widely, and that we want to teach our students. Note that the first three are validation techniques, while the remaining three are “verification” techniques that cross-check the specification against informally-developed program code during execution.

1. **Using animation to validate the specification.** This allows the software engineer (or the client in some circumstances) to interact with the animated specification. Of course, not all specifications can or should be animated [6]. However, specifications that specify output values as an expression over input values can be animated, albeit on small data sets and very slowly. Probably the majority of specifications are written in this form, but there are many exceptions too. Nevertheless, this is a useful way of exploring the functionality of systems. Even if a specified operation does not specify unique outputs, it can sometimes still be animated by prompting the user to supply possible output values (or exhaustively trying all output values, if the state space is very small), then checking these values against the specification.

2. **Using test cases to validate the specification.** The specification of every operation should include examples of acceptable input/output values. These should be automatically checked against the specification each time the specification is modified. For example, given a Z schema \( OP \) with inputs \( i? : T_i \), outputs \( o! : T_o \) and state variables \( s : T_s \) and \( s' : T_s' \), with test values of \( a, b, c, d \) respectively, we simply check that the following is true:

\[
(\exists i? : T_i, o! : T_o, s, s' : T_s \mid i? = a \land o! = b \land s = c \land s' = d \rightarrow OP)
\]

For the majority of specifications, formulae like this can be evaluated to true or false by a good theorem prover, such as Z/EVES [15]. In fact, Z/EVES provides a substitution operator that abbreviates the above to:

\[
OP[i? := a, o! := b, s := c, s' := d]
\]

It is also possible to provide counter-examples for operations, and check that they evaluate to false. The use of positive and negative test cases allows the behaviour of the operation to be bounded on both sides (overspecification, underspecification) and is a useful way of checking that the operation specifies what was intended.

3. **Validating specifications by inspection.**

Code inspection is an important software engineering technique, so we apply it to formal specifications. This provides a good opportunity to teach the usual team member roles and inspection processes, as well as being a good exercise in critiquing specification style.

4. **Generating testsets from formal specifications.** Systematic unit and system testing is one of the primary techniques used to “verify” software correctness in traditional software engineering [13, 5]. Testing can account for a large proportion of the total effort in the software lifecycle, up to 50%. When a formal specification is available, research has shown that it provides
a good guideline for producing comprehensive black-box test sets [2, 4]. A formal specification can also often be used to solve the *oracle problem* in testing – determining if the output of a test run is correct or incorrect [14]. We decided to teach testset generation using the *Test Template Framework* (TTF) of Stocks and Carrington [19, 16, 2]. For each operation specified in Z, the TTF suggests that you build a tree of Z schemas, where the root of the tree is the valid input space of the operation, the nodes of the tree are subsets of that input space and the leaves of the tree correspond to individual tests of the operation. Standard heuristics, such as boundary-value analysis and cause-effect analysis, can be used to design the tree, and the TTF provides several additional heuristics that are guided by the structure of the Z specification.

We made students develop TTF trees by hand, so that the connection between the specification, the heuristics and the resulting TTF tree and testset was clear. However, in real applications, tool support would be useful. Indeed, commercial tools are becoming available to partially automate some of this process [18].

Testing using formal specifications looks to be an exciting and fruitful area for further research and tool development, because it promises to use formal specifications to generate comprehensive testsets more cost-effectively and solve the oracle problem. This in itself would be a powerful reason for writing formal specifications.

5. **Transforming formal specifications into assertions within code.**

*Assertions* are widely recognised as an important technique for improving the quality of code [10] and catching errors during debugging and testing [22]. The basic idea is to insert assertion statements that check the correctness of data structures and algorithms and detect illegal conditions as soon as they arise. Whenever the program is executed, these assertions monitor correctness and raise errors whenever an assertion fails. Typically, the assertions can be disabled via a compiler switch, so that they impose no overhead on production programs.

A formal specification of a procedure is a useful basis for generating assertions. The postcondition of the specification generates assertions at the end of the procedure, to check the correctness of the algorithm within the procedure. The precondition of the specification generates assertions at the beginning of the procedure, to check that clients have called the procedure correctly.

Assertions that are generated from pre/postconditions are typically much more useful for catching errors than assertions that just check for runtime errors like null pointer dereferencing and array overflows (which are more commonly written by programmers). However, the two kinds of assertions are complementary, and we teach both.

Again, an automated or partially-automated system for translating specifications into assertions within code would be a useful tool for catching errors within implementations and another powerful reason for writing formal specifications.
6. **Using formal specifications to specify components of systems such as the classes of an object-oriented system.**

Formal specification is often presented as if it is normally used to specify the behaviour of a complete application. However, it can equally well be used to specify the behaviour of small subsystems (*components*), such as a single C++ class.

A major attraction of component-based specification is that it is not necessary to specify the whole system to gain the most of the benefits of formal specification. Once one component is specified, that specification can be used to generate test sets and assertions etc. for that component. So, component-based specification is cheap to apply, and its benefits are immediate. Furthermore, it is often easier for beginner specifiers to specify a single component than a whole system, because the component is smaller and more concrete and managing complexity and finding good abstractions are less critical for small systems than for large ones.

Another attraction is that it is useful to partially specify a component. For example, a partial specification of a class might involve simply defining a class invariant that checks numeric ranges more precisely than C++ does (e.g., 1...10 rather than just *char*). This still provides some useful documentation and self-checking capabilities for that class.

At the other extreme, the full functionality may be specified, then used to perform comprehensive assertion checking by adding abstract data fields to the class (perhaps using Z-like data types from a library), updating them in parallel with the concrete data fields and checking a *retrieve* function [12] between the abstract and concrete data fields after every method. Like techniques 2 and 4 above, this approach has the advantage that the concrete implementation calculates the new concrete state and output values, so the specification can be used simply as an oracle (checking that the initial and final abstract states satisfy the specification), rather than having to be executable in order to compute outputs itself.

Finally, teaching specification of classes in object-oriented languages is particularly attractive, because it interacts nicely with subtyping (*interfaces* in Java [1]). For example, typical correctness rules for subtyping are similar to the rules for data refinement of an ADT, except that the subtype may add extra data variables and methods [21]. Thus teaching class-based specification gives natural opportunities to introduce or reinforce concepts like data refinement, invariants and programming by contract [11].

Because of these attractions (particularly the fact that it can be applied partially and to any part of a system), we believe that specification of components, rather than whole systems, may become one of the main applications of formal specification in the future. Two promising examples of such an approach are the Extended Static Checkers for Modula-3 and Java [9] and the Java Modelling Language [8], which both add specification constructs to Java and provide various ways of using them to check the correctness of the Java code.
Those are some of the main techniques that we included in the course. Note that they all centre around formal specification, and that none of them require formal verification of code, or extensive knowledge of theorem proving. The only theorem proving that we need is an evaluator for Z expressions and predicates with concrete values given for each variable. This evaluation can often be fully automated.

We decided to use the Z/EVES [15] theorem prover throughout the course, because it performs Z typechecking, can also be used to expand schema expressions and calculate preconditions, and it has a powerful semi-automatic prover that is capable of evaluating most concrete Z expressions automatically.

For work involving programming, we used the C++ language [20] because the students are familiar with it, it is still the standard object-oriented programming language in industry, and because it is one of the most formal-methods hostile languages in common use. This allows us to show students that the techniques we teach can be applied to any language, not just one with clean semantics like Dijkstra’s guarded command language [3].

Our overall goal was to expose students to formal specification techniques that are leading edge yet practical, and at the same time reinforce traditional software engineering skills. This ensures that even those students who will never use formal specifications again will go away from the course with improved practical skills in areas like: safety-analysis techniques; black-box testing; inspection teams; using assertions effectively; rules for object-oriented subtyping; plus an understanding of how formal specifications can improve the effectiveness and precision of these practical areas.

3 Course Outline

This section briefly describes our course design. The course was divided into six modules, each running for two weeks and assessed via a practical assignment (except Module 2, which was assessed via a test).

The six modules were:

Module 1: Computers that Kill. This module covers techniques for measuring and improving the safety of computer-based systems, including risk analysis, hazard analysis and various software engineering techniques for improving quality.

Module 2: What does it do? (Reading Z specifications).

An introduction to formal specifications and the Z specification language.

Module 3: Test, Test and Test Again. This module covers animation of specifications; validation of specifications against test data using Z/EVES; an overview of testing theory; and black-box techniques for generating testsets from formal specifications (using the Test Template Framework).

Module 4: What do I want it to do? (Writing Z specifications).

How to write Z specifications, and check them using Z/EVES. Inspection techniques for specifications.
Module 5: From Specs to Code and Back Again. An introduction to the relationships between specifications and code, based on Hoare logic. The use of assertions to detect errors in programs. Generating assertions from specifications. Theory of data refinement and techniques for handling data refinement within assertions.


In addition to the six assignments (8% each), and a final exam (34%), there was a specification project (18%) that students did in pairs. We asked them to write a formal specification of a Cardax system for controlling doors via swipecards. This was interesting, because they had worked on informal requirements, designs and implementations of a Cardax system in a previous course, so already had a good understanding of the system.

4 Two Edged Assignments

In this section, we describe the two assignments that related most closely to current industry practice (testing and assertions). Each practical was double-edged, in that it developed formal methods skills at the same time as exercising practical software engineering skills.

4.1 Assignment 3: Animation and Testing

The students were given a Z specification that specified two classes (ADTs), called IDSet and GradeSys. The IDSet class simply stored a finite set of student identity numbers. All its operations were deterministic and capable of being animated. The GradeSys class used five instances of the IDSet class to record student grades A to E. This was a good opportunity to illustrate lifting (or promotion) schemas in Z.

The assignment had three parts. Here is a summary of the instructions given to students:

1. Validation by Animation [20 marks].
   
   Choose a sequence of IDSet operations that exercises all of the operations several times, and show how it can be executed by Z/EVES. Your sequence should contain about 10 to 12 operations.

2. Validation by Testing [20 marks].
   
   To show that we can still validate schemas via testing, even when they are not executable, let us define an extra IDSet operation that is non-deterministic. It outputs any one of the students in the given IDSet.

\footnote{Cardax is a trademark of PEC New Zealand Ltd.}
Use Z/ES/S to test each of the public IDSet operations, plus this Someone operation. For each operation, give two positive test cases (that meet the specification) and one negative test case (that contradicts the specification).

3. Testset Generation using TTF [60 marks].
Generate a comprehensive set of black-box tests for an implementation of the student grade system, using the Test Template Framework.
Your tests will be applied to ten implementations of that system. One of the implementations has no known bugs, while the other nine each have one bug inserted. Thirty marks will be given for the number of bugs that your tests detect (3 marks per faulty implementation that you detect). The other thirty marks will be allocated according to how well you have used the TTF and how systematic and well-designed your testset is.
To make things easier for you, and to enable you to judge your progress, six of the implementations (five faulty ones, plus the probably correct one) will be available for downloading from the 424 web site. To keep you in suspense, and to motivate you to develop a comprehensive testset, the remaining four faulty implementations will not be available to you.
In addition to submitting your testset, you must also submit a file called ttf.zed that shows how you developed the testset. This should contain a picture of the TTF tree that you developed, define a Z schema for each node of the tree and explain your reasons for applying particular test generation strategies at each point in the tree.

Note that this practical developed:

- **Formal methods skills:** Z comprehension and writing simple Z schemas.
- **Traditional s/w engineering skills:** Development of black-box testsets.

On average, students submitted about 20 test files, with about 15 commands in each file, and found 7.4 of the 9 errors. As these results suggest, the quality of TTF trees and the resulting testsets was quite good.

4.2 Assignment 5: Assertions

The primary goal of this assignment was for students to learn several ways of adding assertions to C++ classes and to understand the basic concepts of data refinement. Rather than using the standard C++ assert statement, we used several statements: PRE(), POST(), ASSERT() and CLASS_INVARIANT(), to distinguish the different kinds of errors more clearly. Here is a summary of the instructions the students were given:
You are given two C++ classes (containing some errors), and will also have to implement one more class yourself. After you submit the resulting classes, they will be exercised by my special test data (carefully designed to expose the errors) and several client programs that misuse the classes in various ways. If you have done a good job of adding assertions, then any errors that arise will be trapped by your assertions, and you will get top marks. This simulates the real-life scenario of you writing some C++ classes, and adding assertions in order to:

- catch any coding errors within those classes. (In this case, I’ve inserted some non-obvious errors on purpose, so that I know what they are and can mark the assignment fairly across the whole class).
- catch any client programs that use your classes in the wrong way. (I’m pretending to be that stupid client programmer who has not read your class documentation carefully enough, so is writing code that misuses your classes).

The assignment consists of the following tasks:

1. [60 marks] Add extensive assertion checking to the HashSet class (which implements the IDSet Z specification), by adding abstract specification variables to the class, executing the abstract and concrete algorithms in parallel and checking their consistency within the class invariant procedure (which should be called after each method). The retrieve function from the concrete to abstract variables is . . . .

   After doing this, you should have a pretty water-tight system! An extreme case of software redundancy – running two sets of data structures and algorithms in parallel, just like safety-critical systems do for hardware. If both the abstract and concrete subsystems complete execution, and produce results that agree, we can be pretty confident that the answers are correct.

   But note the if they complete execution caveat! The concrete algorithms might still fail due to errors causing nasty C++ runtime errors, such as array bounds overflow or NULL pointer dereferencing. So, it is worth looking briefly through the concrete code that I gave you and adding ASSERT(...) checks anywhere that you think C++ runtime errors might arise. It is better to catch these with assertions than with core dumps!

2. [40 marks] Implement the LogGradeSys class which is specified in grades.zed.

   It is a subtype of the GradeSys system, so in C++, your LogGradeSys class should publicly inherit from the GradeSys class.

   Make sure you override the class invariant, and define the stronger invariant (it is good to call the parent invariant, to avoid duplicating its code). Similarly, you will have to override some methods to alter their functionality as required by the specification, and add the new methods. However, the original GradeSys class did not contain any assertions, so you should also override most of the other methods so that you can add more assertions.

   Unlike part 1 of this assignment, where we went wild with assertions (and duplicated the data structures), in this class I want you to take a more moderate approach.
Try to add assertions that check the precondition of each method. For the postconditions, it is not always convenient to check every detail of the state change and the output results, but try to check a few key aspects of each method. For example, when adding a new student to a grade, perhaps you can check that the size of that grade increases by one. Also, look for any C++ runtime errors that you can catch. The aim is to get a reasonable level of checking, without spending huge amounts of time writing assertions and without adding so many expensive assertions that the program becomes unreasonably slow when run in debugging mode.

Note that this practical developed:

- **Formal methods skills:** understanding the role of preconditions, postconditions, invariants and data refinement.

- **Traditional s/w engineering skills:** robust programming using assertions in C++.

The annotated C++ classes that they submitted were executed under 10 different scenarios; one to check that normal functionality was not compromised by the assertions, three that exercised errors within the *IDSet* implementation (these should be caught by postcondition assertions), three erroneous client calls to *IDSet* methods (should be caught by precondition assertions) and three erroneous client calls to *LogGradeSys* methods (should be caught by precondition assertions).

Most students annotated the *IDSet* implementation well enough to catch all of its three errors (executing the abstract and concrete algorithms in parallel is a stringent test of correctness). On average, they also correctly caught\(^2\) 54% of the *IDSet* client errors and 60% of the *LogGradeSys* client errors.

### 5 Results

We ran the redesigned course in the first semester of 1998, using the same textbook, lecturers and course title as in 1997. As we hoped, student morale and course evaluations improved markedly.

The overall course rating improved from 2.9 to 2.3, with no students rating the course as *unsatisfactory* (4) or *poor* (5) (in 1997, 32% rated it *unsatisfactory* (4)). More telling was that 76% of students would now encourage others to take the course (36% in 1997). A higher percentage of students (47%) stated that their interest in the subject was increased and only one student said his/her interest was reduced.

Practicals 3 and 5 were given the best rating on the *usefulness* scale, which is understandable because they relate most closely to current industry practice (testing and assertions).

\(^2\) By correctly caught, we mean that a PRE() assertion was triggered. They were given reduced marks if some other kind of assertion was triggered.
It was also our informal impression that students were producing better quality Z specifications than students from previous years, in spite of us spending less time teaching Z directly. However, in one exam question that was similar in 1997 and 1998, there was no significant improvement.

We conducted one small explicit experiment on the students (with their knowledge). We set them the exercise of writing two specifications: one for a simple email system; one for a simple hierarchical filing system. About half of the students were also asked to perform tests (along the lines of Technique 2 in Section 2) on the email system, the other half of the students were asked to do the same on the file system. We hoped to detect some, purely qualitative, difference between the specifications that were and were not tested.

During marking of this work two things became apparent. One was that even when testing showed that there was something wrong with the specification, the students did not seem to use this as a prompt to go back to the specification and repair it. Instead, they gave the results of the tests, perhaps mentioned that the test had failed, and moved on.

A variant of this behaviour seemed to be caused by wishful thinking. In this variant, Z/EVES would typically not have reduced the testing predicate to either true or false, but rather to some combinatorially complex but semantically simple expression. Having at the back of their minds the expected outcome of the test (e.g that the test should have produced ‘true’) the students typically stated that the complex expression ‘obviously’ or ‘clearly’ (always a danger signal when those words are used!) evaluated to the expect truth-value even when, in fact, it evaluated to the opposite truth-value.

The other commonly observed problem with the testing was that the students used special cases for the tests, which were misleading as to the correctness of the specifications. The most usual special case was to use empty sets in the state upon which an operation was used. Particularly in cases where some predicate involved quantified statements over the sets concerned, errors in the predicate were not made apparent by the test since an empty set typically made the predicate vacuously hold when it would not have done in a general case. Also, of course, an empty set simply cannot cover as many cases of how the state might look, so erroneous parts of predicates are less likely to be ‘exercised’. This problem seems to be caused in part by laziness—it is simply easier to type in an empty set than a non-empty one.

At least, having experienced these common problems, it will be easy to alert students to them in future—ensuring that, nevertheless, they do not succumb will, of course, be much harder.

Here is a typical sampling of students comments about the whole course, as given on the anonymous course survey forms:

Interesting concepts introduced. NEW IDEAS, possibly some ideas that aren’t used in the industry yet are taught in this course. Good thing, but possibly a bad thing also.

Course overall was good. The testing and assertions sections are relevant to getting a good job and maybe Z will be in the future as well.
Improvement over what I’ve seen from last year. Much more varied and less theoretically based.

Boring in parts.

I think the course is well structured. The course contents are interesting.

In response to the question “How useful do you think your experience from this course will be in your future career?”, some student answers were:

Extremely useful. My understanding of testing is improved. Writing more solid code has had an impact on all of my work this year and I’m sure it will continue to improve the standard of my work.

Very. Gives me a good understanding of how programs (especially large and complex ones) should be developed.

A lot!

Of course, a few students were less convinced:

Actual application may be minimal, but at least I can say I know the basics of formal methods.

One student wrote as he submitted Assignment 3 (Testset generation from Z specifications):

I found this assignment challenging and exciting. It did take a long time and even now I am not sure if I sent you my most up to date or correct files (I have test cases everywhere) but it has helped me to understand Z schemas, Z-EVES and the idea of testing. While I don’t think I could ever be a tester for large systems, I found testing programs in this structured way interesting and at times captivating (as I tried to hunt down errors).

6 Conclusion

We have shown that incorporating several novel techniques that use formal specifications in various parts of the software lifecycle led to a measurable improvement in our formal methods course, including increased student satisfaction, and much higher ratings on usefulness.

We believe that the six techniques we have explored need to be more widely used in industry generally as well. We hope to trial some of them in a formal specification project that we have just started, collaborating with several New Zealand software developers. We urge other researchers and tool developers to begin or continue working in these important areas.

---

3 This project is supported by the New Zealand Foundation for Research, Science and Technology (FRST).
In future years, we want to explore more sophisticated tool support for some modules. We plan to evaluate Zylva [18] for the testset generation practical, and perhaps use JML [8] as a specification language for object-oriented components. We would also like to include some material on model-checking, because it is an important and practical technique for state machine systems and has good tool support. However, finding room for all the great topics we would like to teach is not always possible.

References


Issues for a Temporal Refinement Calculus

Ron van der Meyden\textsuperscript{1} and Yoram Moses\textsuperscript{2}

\textsuperscript{1} Computing Sciences
University of Technology, Sydney
P.O. Box 123, Broadway
NSW 2007, Australia
ron@soc.scs.uts.edu.au
http://www-staff.socs.uts.edu.au/~ron

\textsuperscript{2} Electrical Engineering
Technion, 32000
Israel
moses@ee.technion.ac.il

Abstract. This note discusses a number of issues that arise in the development of a refinement calculus for a programming language in which specification statements may involve temporal logic statements. We consider a sequential (i.e., non-distributed) setting and aim at a linear-time semantics for the temporal logic. The added power due to the use of temporal operators is illustrated, and subtleties having to do with proving assertions about the program and the use of labels in a program are considered. This work is part of a project whose goal is to provide a refinement calculus for distributed programs using a rich specification language.

1 Introduction

We are engaged in a project\textsuperscript{1} to develop a formal framework for the design of distributed algorithms. Our starting point in this work was an approach to the analysis of such algorithms that is based on logics of knowledge [8], which has been shown in a number of works in the last decade to provide unique insights into the nature of distributed computation. The knowledge-based programs of Fagin et al [5] provide a formal syntax and semantics within which this approach can be represented. Knowledge-based programs resemble ordinary programs in their syntax, but they are better viewed as constituting abstract specifications of the way a process’ concrete actions relate to its state of knowledge. Knowledge-oriented programs [11] take the level of abstraction one step further, allowing abstract actions such as notify whose purpose is to effect changes in processes’ states of knowledge.

We aim to generalize the frameworks of knowledge-based and knowledge-oriented programs. Inasmuch as the objects of study in these frameworks are a

\textsuperscript{1} Supported by an ARC Large grant and by a grant from the Israeli Ministry of Science.
hybrid of programs and specifications, they resemble refinement calculi, which similarly view programs and specifications as belonging to the same syntactic and semantic category. This analogy has led us to the investigation of refinement calculi for distributed systems that are capable of representing knowledge-based programs. The work has required us to generalize and amend previous approaches to program refinement along several dimensions. Specifications of and reasoning about distributed computation deal inherently with temporal behaviour. Thus, one of the issues we have found it necessary to consider is how to integrate program refinement and temporal logic. The purpose of this note is to briefly describe our approach to such an integration (in the context of non-distributed programs, for which the problem is already quite complex) and some of the issues we have had to face. Details will be provided in a forthcoming paper [12]. Other aspects of our larger program of research are treated in [4, 13].

Previous work on program refinement can be categorised into two classes. One of the most deeply explored approaches [15, 3] is state-based, being premised on the use of predicate transformers and weakest preconditions as a semantic basis. This is natural for programs whose specifications are descriptions of input/output relations, of which predicate transformers are a generalization. One of the advantages of this approach is the ability to write specifications containing two distinct types of annotation: one type of annotation is used to state properties that the program is required to satisfy, the other is more like conventional program annotations in that it is used to state properties that the program has been proved to satisfy.

There have been proposals to extend the application of the predicate transformer based approach to distributed computations, by viewing a distributed program as a predicate transformer which is repeatedly applied [1, 2]. (There are also closely related proposals to base the semantics of temporal logic on predicate transformers [16, 10].) A disadvantage of this idea is that the rich compositional structure of the standard refinement calculus is lost.

Another class of refinement calculi is action-based [6, 9, 7, 19]. Work in this category begins with a process calculus and enriches it with features of modal logic, typically Hennessy-Milner logic or the more expressive $\mu$-calculus. This approach has the benefit of allowing greater compositional freedom. On the other hand, these works do not admit the two types of annotation available in the state-based approaches.

The contribution of our work can be viewed as being to develop a framework that has both the rich compositional structure and modal logic expressiveness of the process calculus based approaches and the ability to represent the two types of annotation available in the predicate transformer based approaches to refinement of sequential programs. Instead of an action based modal logic, we use temporal logic. In Section 2 we sketch a program calculus based on temporal constraints. Section 3 extends this calculus with assertions that capture annotations by proved properties. Section 4 deals with a further useful enhancement, the use of labels that may be referred to by the annotations.
2 Programs with Temporal Constraints

Our approach is grounded in the standard semantics of temporal logic [17]. Suppose for the sake of discussion that we are interested in a propositional temporal logic in which for every formula \( \varphi \) there is a formula \( \Diamond \varphi \), intuitively asserting that \( \varphi \) will eventually be true. The basic semantic construct in the standard semantics is a run, a mapping from timepoints (the natural numbers \( \mathbb{N} \), say) to states of the system under consideration. A point is a tuple \((r, n)\) where \( r \) is a run and \( n \) is a natural number. We may define the relation \( \models \) of satisfaction of a formula at a point by

1. when \( p \) is an atomic proposition, \( (r, n) \models p \) if \( p \) holds in the state \( r(n) \),
2. \( (r, n) \models \Diamond \varphi \) if \( (r, m) \models \varphi \) for some \( m \geq n \).

This is the linear time semantics in the style of Manna and Pnueli [17]. As usual, we write \( \Box \varphi \) for the dual \( \neg \Diamond \neg \varphi \), which asserts that \( \varphi \) is true at all future times.

In order to develop a theory of program refinement, we start by defining three elements:

- A language of programs.
- An executes relation \( \models \) between intervals of runs and programs. An interval is a triple \( r[n, m] \) where \( r \) is a run and \( n \leq m \leq \infty \) are elements of \( \mathbb{N} \cup \{\infty\} \).
  Intuitively, \( r[n, m] \models P \) holds when the program \( P \) executes from time \( n \) to time \( m \) in the run \( r \).
- A refinement relation \( \sqsubseteq \), where we write \( P \sqsubseteq Q \) for program \( P \) refines program \( Q \).

Our language of programs contains analogues of the usual constructs for sequential programs. Chief amongst these is an operator \( * \) which resembles sequential composition. Its semantics is given by \( r[m, n] \models P * Q \) if there exists \( m' \) such that \( r[m, m'] \models P \) and \( r[m', n] \models Q \).

In addition to standard operators of a sequential language, our language of programs contains two types of basic programs that are thought of as constraints, which are in the spirit of similar constructs used by Morgan [15] and by Back and von Wright [3]. The first are the so-called coercions, which have the form \( [\varphi] \) and force \( \varphi \) to hold. The second are specifications, which have the form \( [\varphi, \psi] \), standing for a program that, if started at a point satisfying (the initial condition) \( \varphi \), will terminate in a point satisfying (the goal) \( \psi \). Roughly speaking, we want the \( \models \) relation for such program constructs to satisfy

- \( r[n, m] \models [\varphi] \) if \( n = m \) and, if \( n < \infty \), then \( (r, n) \models \varphi \); and
- \( r[n, m] \models [\varphi, \psi] \) if either
  1. \( n = m = \infty \) or
  2. \( n < \infty \) and \( (r, n) \not\models \varphi \) or
  3. \( n \leq m < \infty \) and \( (r, n) \models \varphi \) and \( (r, m) \models \psi \).

2 We use the reverse of the order preferred by many authors, for reasons having to do with correspondence to our semantics of this relation.
By being instances of basic programs, constraints can be composed with other constructs, resulting in composite programs that can have elements which are programs at different levels of abstraction.

Allowing temporal operators in the formulas enables the coercions and specifications to talk about ongoing and reactive behaviours. This gives our framework an expressive power that goes beyond input/output relations. For example, we may write programs such as

$$[\square(x < 10)] * [\Diamond(x = 3), x = 2]$$

This specifies a program that ensures that the value of $x$ is always less than 10, and if started in a situation where it is guaranteed that eventually $x$ equals 3, will terminate with $x$ equal to 2.

Given the elements described above, we can define a refinement relation on programs by $P \subseteq Q$ if for every $r[n, m]$, if $r[n, m] \models P$ then $r[n, m] \models Q$. This captures the intuition that refinement corresponds to the addition of constraints to the program, and the reduction of any nondeterministic behaviour it may allow. While this definition is intuitive, however, we have not yet captured some important features of the standard refinement calculi, that make a significant contribution to their practicality.

3 \ Assertions

It is often very useful to be able to reason about a partially refined program, some of whose components may still involve constraints. Properties guaranteed by certain parts of the program can then be made use of to improve the refinement of other parts of the program. This becomes especially useful once our formulas involve temporal operators, because then it may be possible to infer temporal properties at one point from those that are proven to hold at another. We call such conclusions about the program assertions, and denote the assertion that $\varphi$ holds by $\{\varphi\}$. Syntactically, assertions are very similar to coercions. Semantically, however, they are treated differently.

One of the features we wish to make use of is the fact that assertions may be stated based on constraints in the program, which we think of as program fragments that will be developed later in the refinement process. For example, at a location in the program where a coercion $[\varphi]$ appears, we may soundly assert that $\varphi$ must hold. Every concrete program that refines the current text will indeed satisfy $\varphi$ at that location. Conversely, one way to demonstrate that a coercion has been satisfied is by proving that the formula being coerced is a sound assertion there. Done properly, this should be a way of eliminating coercions in the refinement process.

The generation and use of assertions involves a number of subtle points. First, it is essential that the two operations mentioned above: introducing an assertion based on a coercion, and eliminating a coercion based on an assertion, are not applied inappropriately, as in the following derivation:

$$\epsilon \subseteq \{\varphi\} \subseteq \{\varphi\} * [\varphi] \subseteq [\varphi]$$
Here $\epsilon$ stands for the empty program which does nothing and takes no time to execute. The derivation should be read from right to left. In its first step, the assertion $\{\varphi\}$ is concluded from the coercion $[\varphi]$. In the second step, the same assertion is used as basis for eliminating the coercion(!) The third step eliminates the assertion, which is a step that is consistent with our view of assertions as stating properties that have already been established, and therefore not constraining the refinement process. The problem in this example is, of course, the circularity of the reasoning. Similar issues can arise in much more subtle ways, especially in the presence of temporal formulas. For example,

$$
\epsilon \subseteq [\diamond p, \diamond p] \subseteq \{\diamond p\} \ast [\text{true}, \diamond p] \subseteq [\text{true}, \diamond p] \ast \{\diamond p\} \subseteq [\text{true}, \diamond p]
$$

is a problematic derivation due to a similar circularity.

In order to avoid such pitfalls, our formalism is enhanced to keep track of the constraints on which the statement of an assertion is based. Technically, we label constraints with constraint variables from a set CV, so that a coercion, say, will now have the form $[\varphi]^X$ where $X \in CV$, and every assertion appears as $\{\psi\}_J$ where $J \subseteq CV$ is considered the justification set for this assertion. Intuitively, an occurrence of $\{\varphi\}_J$ at a location in a program states that $\varphi$ holds at this location, with the proof depending only on constraints labelled by variables in $J$. We now allow the introduction of assertions as in

$$
\{\varphi\}_J \ast [\varphi]^X \subseteq [\varphi]^X.
$$

We also have rules for the elimination of constraints, such as

$$
\epsilon \subseteq \{\varphi\}_J \ast [\varphi]^X \quad \text{if } X \notin J.
$$

The restriction in this rule suffices to block the first problematic derivation above, and similar restrictions can guarantee to block the second one. Properly keeping track of the reasoning underlying the assertions in this way requires, however, a somewhat more complex semantics for both programs and refinement. We need to guarantee that refinement steps generate correct assertions, and may need to modify justification sets in the course of a refinement step. The definition of refinement we use is a generalization of that of the previous section. The details are beyond the scope of this brief abstract (but see [13]).

4 Labels

In order to translate reasoning between the program constructs and the temporal logic, we have found it useful to introduce labels as a way of marking specific locations in a program to which the temporal assertions may refer. Intuitively, it may be surprising at first to see the assertion placed before the assertion, when intuitively it is “caused” by the coercion. This is not an error, however, since the assertion and the coercion refer to the same point of the run, and therefore commute.
to each label $l$ there exists an atomic proposition $\text{at}(l)$ true just at the locations marked by $l$ in the program. Labelled programs and reasoning using the propositions $\text{at}(l)$ are a central feature of the approach to temporal verification of Manna and Pnueli [17, 18]. In this approach, the whole program needs to be translated to a formula of temporal logic. The incorporation of labels in our framework makes it possible for us to simulate this approach. However, because we have a richer syntax, we have greater flexibility to do reasoning based directly on the structure of the program, without having to translate it to temporal logic in its entirety.

Labels also add to the expressive power of our refinement calculus in interesting ways. For example, provided the label $l$ does not occur in $P$,

$$\exists l[\text{at}(l) * P * l)$$

defines a program whose behaviours are precisely the terminating behaviours of the program $P$. Here the existential quantifier functions as a scoping (or hiding) operator whose purpose is to delimit the points of the run at which $l$ occurs to those referred to by the program in its scope.

5 Conclusion

The framework we have sketched above is able to capture much of the reasoning that can be represented in refinement calculi for sequential programs based on predicate transformers, but not all. Restricting our framework to sequential programs, with assertions restricted to be about states, would make its semantic basis be roughly equivalent to the set of relations over a given collection of states. The set of predicate transformers forms a much larger semantic space. This allows standard refinement calculi to represent “miraculous” programs which, while they cannot correspond to any concrete program as representable in a conventional programming language, can nevertheless be used during program derivation [14]. Our framework does not admit such miraculous programs. Since we do have the capacity to handle standard temporal reasoning in our framework, we believe the price is one worth paying.

Acknowledgements

Thanks to Kai Engelhardt for comments on a draft of this paper.
References

Author Index

Butler, Michael .................. 1
Carrington, David ............... 58, 97
Dunwoody, James ................. 129
Hartel, Pieter .................... 1
Harwood, Keith ................... 13
Hayashi, Yuji ..................... 22
Ibrahim, Rosziati ................. 40
MacleColl, Ian .................... 58
Miyamoto, Eiichi ................. 22
Moses, Yoram .................... 159
Qiu, Winnie Weiqun ............. 70
Reeves, Steve .................... 145
Schwenke, Martin ................. 85
Shield, Jamie ..................... 97
Szyperski, Clemens .............. 40
Tarasyuk, Igor V. ................. 110
Utting, Mark ..................... 129, 145
van der Meyden, Ron ............ 159
Yamaguchi, Tadashi .............. 22
Žic, John ......................... 70