Specifications of Digital Rights Languages

Y. Arimoto, D. Bjørner, X. Chen, J. Xian
{arimoto,y,xychen,bjorner,jxiang}@jaist.ac.jp
The Digital Rights: Consumers and Producers in a Digital World Subgroup
Domain Engineering and Digital Rights Group
School of Information Science

1-1, Asahidai, Tatsunokuchi,
Nomi, Ishikawa, Japan 923-1292

June 29, 2006
Abstract

We present a series of specifications of digital rights concepts and languages for digital rights management.

The specification are alternative to specifications given in a number of seminal papers. Each such paper is reviewed, i.e., an extensive transcription is presented and comments are made about problems that we encountered when studying the paper. Then we bring in a specification expressed in the RAISE [1] specification language RSL [2]. And thirdly we bring in a specification in the algebraic specification language CafeOBJ [3, 4]. We finally compare the three specification approaches: those of the published papers, the RSL and the CafeOBJ approaches.

The aims of this document is to understand. We do so by capturing our understanding in three ways.

The objective of this document is to present, to a wider audience, in a comprehensive form some fundamental issues of digital rights management.

---

A Book Proposal

One of the coauthors of this report suggests editing the report into a small book to be published under the working title: Digital Rights: Languages and Management and with the subtitle Consumers and Producers in a Digital World. If this proposal is feasible (within a January 2007 deadline) it is furthermore suggested that the authorship list be extended with Kokichi Futatsugi, Kazuhiro Ogata and René Vestergaard — provided, of course, that they too will contribute to the book. The book is estimated to be some 180 pages including indexes and appendices.
Contents

Abstract 3
A Book Proposal 3

1 Introduction 9

2 The Gunter/Weeks/Wright’s Paper 11
2.1 Transcript of the Gunter/Weeks/Wright Paper 11
2.1.1 Actions, Events, Realities and Licenses 11
2.1.2 Standard Licenses 14
Up Front Licenses 14
Flat Rate Licenses 14
Per Use Licenses 15
Up to Expiry Date Licenses 15
Non-cancellable Multi-use Licenses 15
Cancellable Multi-use Licenses 16
2.1.3 A License Language 17
Syntax 17
Example 17
Semantics 17
2.1.4 Comments 17
2.2 An RSL Model 17
2.2.1 Actions, Events, Realities and Licences 18
2.2.2 Standard Licences 19
Syntax 19
Semantics 20
2.2.3 A License Language 23
2.3 A CafeOBJ Model 23
2.3.1 CafeOBJ Specification 23
2.3.2 Discussion 31
2.4 Comparisons 32

3 The Pucella/Weissman Paper 33
3.1 A Transcript of the Pucella/Weissman Paper 33
3.1.1 The Logic $L_c$ 33
Syntax 33
Example 34
Semantic 34
3.1.2 Encoding finite runs and licenses 36
3.1.3 Handling different license languages 36
3.1.4 Comments 36
3.2 A RAISE/RSL Specification 36
3.2.1 Three Sublanguages 36
3.2.2 The Action Command and Event Command Languages 37
The Action Language ........................................... 37
The Event Language ........................................... 37
Event Complements ............................................... 37
3.2.3 The License Language .................................... 38
3.2.4 The Modal Language of Propositions .................... 38
3.2.5 The “Meaning” of Licences ................................ 40
3.2.6 The Two “Player” License/Action “Game” .................. 42
  Configurations ............................................... 42
  Runs ....................................................... 42
  License and Action Observers on Runs ......................... 43
  Specifically Licensed Actions over Runs ....................... 43
  Permission Observer on Runs .................................. 44
3.2.7 The Proof System ......................................... 45
  Meaning of Action Expression .................................. 45
  Proof Rules .................................................. 45
  Properties of the Proof System ................................ 45
3.2.8 Discussion ............................................... 45
3.3 A CafeOBJ Specification ..................................... 45
3.4 Comparisons ................................................ 47

4 The ODRL Paper ............................................... 49
4.1 A Transcript of the ODRL Paper ................................. 49
  4.1.1 Motivation .............................................. 49
    Remove Ambiguity of ODRL .................................. 49
    Formal Reasoning of Licenses ................................. 50
  4.1.2 The ODRL Language ..................................... 50
    Example of ODRL Agreement .................................. 50
    Abstract syntax for ODRL ................................... 51
  4.1.3 A Semantics in First-Order Logic ......................... 51
    Vocabulary ............................................... 51
    Translation Models ........................................ 52
  4.1.4 Queries ................................................ 53
  4.1.5 Summary ............................................... 54
4.2 A RAISE/RSL Specification ................................... 55
4.3 A CafeOBJ Specification ..................................... 55
4.4 Comparisons ................................................ 55

5 The XrML Paper ................................................. 67

6 The DRM Reference Model Paper .................................. 69

7 Digital Rights and Game Theory ................................ 71
  7.1 Preventing Secrets Sharing in Online Subscription Services .... 71
    7.1.1 Problem Domain ....................................... 71
    Problem ................................................... 71
    Technical Solutions ........................................ 72
    Legal Measures ............................................. 72
# User’s Concerns

## 7.1.2 Basic Scheme of IBSPS

## 7.1.3 Details of IBSPS

## 7.1.4 Possible Advantages of IBSPS

## 7.1.5 Discussion

# Conclusion

# Bibliographical Notes

## A RSL: The RAISE Specification Language

### A.1 Types

#### A.1.1 Type Expressions

#### A.1.2 Type Definitions

- Concrete Types:
- Subtypes
- Sorts (Abstract Types)

### A.2 The RSL Predicate Calculus

#### A.2.1 Propositional Expressions

#### A.2.2 Simple Predicate Expressions

#### A.2.3 Quantified Expressions

### A.3 Concrete RSL Types

#### A.3.1 Set Enumerations

#### A.3.2 Cartesian Enumerations

#### A.3.3 List Enumerations

#### A.3.4 Map Enumerations

#### A.3.5 Set Operations

#### A.3.6 Cartesian Operations

#### A.3.7 List Operations

#### A.3.8 Map Operations

### A.4 \(\lambda\)-Calculus and Functions

#### A.4.1 The \(\lambda\)-Calculus Syntax

#### A.4.2 Free and Bound Variables

#### A.4.3 Substitution

#### A.4.4 \(\alpha\)-Renaming and \(\beta\)-Reduction

#### A.4.5 Function Signatures

#### A.4.6 Function Definitions

### A.5 Further Applicative Expressions

#### A.5.1 Let Expressions

#### A.5.2 Conditionals

#### A.5.3 Operator/Operand Expressions

### A.6 Imperative Constructs

#### A.6.1 Variables and Assignment

#### A.6.2 Statement Sequences and \textbf{skip}

#### A.6.3 Imperative Conditionals

#### A.6.4 Iterative Conditionals

#### A.6.5 Iterative Sequencing
Chapter 1

Introduction
2.1 Transcript of the Gunter/Weeks/Wright Paper

This transcript was kindly provided by the first author. It has been slightly edited by the second author. More editing work will be done.

2.1.1 Actions, Events, Realities and Licenses

The model centers around a domain of realities and a domain of licenses.

- A reality is a sequence of events.
- A license is a set of realities.

The semantics of a rights management languages can be expressed as a function that maps terms of the language to elements of their domain of licenses.

Their abstract model represents an event, \( e \in \text{Event} \), as a pair of a time, \( t \in \text{Time} \), and an action, \( a \in \text{Action} \):

\[
e ::= t : a
\]

\( \text{Time} \) is totally ordered by \(<\). The function \( + : \text{Time} \times \text{Period} \rightarrow \text{Time} \) adds a period, \( p \in \text{Period} \), to a time.

There are two kinds of actions:

\[
a ::= \text{render}[w, d] \mid \text{pay}[x]
\]

\( w \in \text{Work} \) denotes a rights-managed work. \( d \in \text{Device} \) represents a DRM-enabled device. \( x \) is a decimal. Action \( \text{render}[w, d] \) represents rendering of work \( w \) by rights-enabled device \( d \). Action \( \text{pay}[x] \) represents a payment of amount \( x \) of some currency from a licensee to a license issuer.

The event \( t : \text{render}[w, d] \) means that at time \( t \), work \( w \) was rendered on device \( d \). The event \( t : \text{pay}[x] \) means that at time \( t \), a payment of amount \( x \) was made.

A reality, \( r \in \text{Reality} \), is a finite set of events, such that all events occur at distinct times:

\[
\text{Reality} = \{ E | \text{Event-set} \cdot t : a \in E \land t : a' \in E \Rightarrow a = a' \}
\]
where $P(x)$ represents the powerset of $E$ (the set of all subsets of $E$). Notation $r_{\leq t}$ represents the prefix of $r$ that occurs at or before time $t$; that is: $r_{\leq t} = \{t' : a \in r \mid t' \leq t\}$. Notation $r \subseteq r'$ indicates that $r$ is a prefix of $r'$; that is, there exists a $t$ such that $r = r'_{\leq t}$.

### Problems

- $r \subseteq r'$ can be simply interpreted as $r$ is a subset of $r'$, i.e., all the actions in $r$ must also exist in $r'$?

In the model, a license, $l \in License$, is a set of realities:

$$l \in License = P(Reality)$$

Let us take an example:

$$l_A = \begin{cases} 
\{8 : 00 : \text{pay}[$1$], 8 : 01 : \text{render}[w_1, d_1]\}, \\
\{8 : 00 : \text{pay}[$1$], 8 : 02 : \text{render}[w_1, d_1]\}, \\
\{8 : 00 : \text{pay}[$1$], 8 : 03 : \text{render}[w_1, d_1]\}, \\
\{8 : 00 : \text{pay}[$1$] \}
\end{cases}$$

### Problems

- The definition of license treats both render and pay as the same obligated actions, that is, whenever the action render occurs in a reality of a license, it means that the consumer MUST render the work otherwise he or she will breach the license. For instance, the third reality of the above example states that the work $w_1$ must be rendered on the device $d_1$ at time 8:03 and 8:04 rather than that it is allowed to be rendered from 8:03 to 8:04!

- This may cause inconvenience when we want to represent a license in which there is no obligation for the consumer to use the service, though the authors state that their definition of license is ODD from a commercial perspective and argues that the licenses with no obligations are UNUSUAL and are usually more complex licenses which are not further discussed in the paper. The question is that, is such license really unusual or common?

- Recent RELs such as ODRL has distinguished these two kinds of actions by introducing the concepts of permission and requirement (a kind of obligation or constraint) with respect to the actions render and pay defined in Gunter’s model.

To give a more formal meaning to a license, suppose $r$ is the (unique) complete reality that actually occurs over the entire lifetime of the DRM system. And let $r[l]$ be events of $r$ attributed to license $l$.

### Definitions
⋆ Reality \( r \in l \) of license \( l \) is viable for \( r[l] \) at \( t \) iff \( r[l]_{\leq t} \sqsubseteq r \).
⋆ License \( l \) is fulfilled by \( r[l] \) at \( t \) iff \( r[l]_{\leq t} \in l \).
⋆ License \( l \) is breached by \( r[l] \) at \( t \) iff there does not exist \( r \in l \) that is viable for \( r[l] \) at \( t \).

**Problems**

• How about \( r[l]_{\leq t} = \emptyset \) as for viable? \( \emptyset \sqsubseteq r? \)

**Example 1:**

\[
\begin{align*}
\mathbf{r}[l_A] = \{ & 8:00: \text{pay}[$1], 8:01: \text{render}[w_1,d_1], 8:05: \text{render}[w_1,d_1] \}.
\end{align*}
\]

For \( \mathbf{r}[l_A] \),

⋆ at \( t < 8:01 \), all four realities of license \( l_A \) are viable
⋆ at \( t \), \( 8:01 \leq t < 8:05 \), only the first reality is viable
⋆ at \( t \geq 8:05 \), no reality is viable

**Example 2:**

\[
\begin{align*}
\mathbf{r}[l_A] = \{ & 8:00: \text{pay}[$1], 8:01: \text{render}[w_1,d_1], 8:05: \text{render}[w_1,d_1] \}.
\end{align*}
\]

License \( l_A \) is

⋆ unfulfilled by \( \mathbf{r}[l_A] \) for \( t < 8:00 \)
⋆ fulfilled for \( 8:00 \leq t < 8:05 \)
⋆ breached for \( t \geq 8:05 \)

**Example 3:**

\[
\begin{align*}
\mathbf{r}'[l_A] = \{ & 8:00: \text{pay}[$1], 8:03: \text{render}[w_1,d_1] \}.
\end{align*}
\]

License \( l_A \) is

⋆ unfulfilled for \( t < 8:00 \)
⋆ fulfilled for \( 8:03 \leq t < 8:03 \)
⋆ unfulfilled for \( 8:03 \leq t < 8:04 \)
⋆ breached for \( t \geq 8:04 \)

**Problems**

• As for \( t < 8:00 \), are the realities of the license viable or not?
2.1.2 Standard Licenses

Up Front Licenses

The “UP Front” license provides access to any work in set $W \in P(Work)$ on any device in set $D \in P(Device)$ beginning at time $t_0$ for period $p$, for an up-front payment of $x$:

$$\text{UpFront}(t_0, x, p, W, D) = \{ t_0 : \text{pay}[x],$$
$$t_1 : \text{render}[w_1, d_1], ..., t_n : \text{render}[w_n, d_n]$$
$$| n \geq 0,$$
$$t_0 < t_1 < ... < t_n < t_0 + p,$$
$$w_1, ..., w_n \in W, d_1, ..., d_n \in D \}$$

**Problems**

- The use of $n$ is confusing.
- On one hand it is used to express up to $n$ renderings.
- On the other hand subscript $n$ is used for ranging both works and devices.
- The three uses of $n$ should be separated into, say, $i, j$ and $k$.
- The same comments apply to the Flat Rate and Per Use formulations.
- Why not allow $t_n \leq t_0 + p$?

**Problems**

- Does the “Up Front” license requires that the consumer must render the work at every specific time $t_1, t_2, ..., t_n$? If so, such license seems strange in practice; if not, how to enumerate all the possible realities such that the license can “provides access to any work in set $W \in P(Work)$ on any device in set $D \in P(Device)$ beginning at time $t_0$ for period $p$”? The same question (i.e., obligation or permission question) occurs as for the following two standard licenses.

Flat Rate Licenses

The “Flat Rate” license provides access to any work in set $W$ on any device in set $D$ beginning at time $t_0$ for period $p$, for a payment of $x$ at the end of the period:

$$\text{FlatRate}(t_0, x, p, W, D) =$$
$$\{ t_1 : \text{render}[w_1, d_1], ..., t_n : \text{render}[w_n, d_n]$$
$$t_{n+1} : \text{pay}[x],$$
$$| n \geq 0,$$
$$t_0 < t_1 < ... < t_n < t_{n+1} < t_0 + p,$$
$$w_1, ..., w_n \in W, d_1, ..., d_n \in D \}$$
 Problems

• Why not allow \( t_n \leq t_0 + p? \)

**Per Use Licenses**

The “Per Use” license is provides access to any work in set \( W \) on any device in set \( D \) beginning at time \( t_0 \) for a period of length \( p \), for a payment of \( x \) per use at the end of the period:

\[
\text{PerUse}(t_0, x, p, W, D) = \{ t_1 : \text{render}[w_1, d_1], ..., t_n : \text{render}[w_n, d_n] \\
\quad t_{n+1} : \text{pay}[nx], \\
\quad | \quad n \geq 0, \\
\quad t_0 < t_1 < ... < t_n < t_{n+1} < t_0 + p, \\
\quad w_1, ..., w_n \in W, d_1, ..., d_n \in D \}
\]

 Problems

• Why not allow \( t_{n+1} \leq t_0 + p? \)

**Up to Expiry Date Licenses**

A license that a consumer can accept any time before some future can be constructed by quantifying over \( t_0 \) for any of the three families defined on preceding slides. For example,

\[
\bigcup_{t_0 < t_{\text{expires}}} \text{UpFront}(t_0, x, p, W, D)
\]

This license allows the period of the use to begin anytime prior to \( t_{\text{expires}} \).

 Problems

• Why not allow \( t_0 \leq t_{\text{expires}}? \)

**Non-cancellable Multi-use Licenses**

To construct multi-period non-cancellable licenses by composing single-period licenses, they define an operator \( \triangle \).

\[
l_1 \triangle l_2 = \{ r_1 \cup r_2 \mid r_1 \in l_1, r_2 \in l_2 \}
\]

where all of the events of \( l_1 \) occur at different times from the events of \( l_2 \).

 Problems

• The above is not the same as: \( l_1 \triangle l_2 = l_1 \cup l_2 \)
• How does $\triangle$ associate?
• We guess: $l \triangle l' \triangle l'' = l \triangle (l' \triangle l'')$

UpFront$^\triangle(t_0, x, p, W, D, m)$ provides access to any work in set $W$ on any device in set $D$ beginning at time $t_0$ for $m$ periods of length $p$, for payments of $x$ at the beginning of each period:

$$
\text{UpFront}^\triangle(t_0, x, p, W, D, m) = \\
\text{UpFront}(t_0, x, p, W, D) \triangle ... \triangle \\
\text{UpFront}(t_{m-1}, x, p, W, D)
$$

where $t_i = t_0 + ip$ for $i$ from 0 to $m - 1$

**Cancelable Multi-use Licenses**

With the $\triangleright$ operator cancelable multi-period licenses can be defined.

$$
l_1 \triangleright l_2 = \{r_1 \cup r_2 \mid r_1 \in l_1, \ r_1 \neq \emptyset, \ r_2 \in l_2\} \cup \{\emptyset\}
$$

where all of the events of $l_1$ occur at different times from the events of $l_2$.

**Problems**

• Same question concerning associativity.
• What, exactly is the rôle of the empty set $\{\}$, or, rather, $\{\{\}\}$ (in Gunter et al. paper: $\{\emptyset\}$).
• We guess: To have only actions from $l_1$ and then “get out”!
• Also: There is a problem with $t_i$: it is defined but never used!

UpFront$^\triangleright(t_0, x, p, W, D, m)$ provides access to any work in set $W$ on any device in set $D$ beginning at time $t_0$ for $m$ periods of length $p$, for payments of $x$ at the beginning of each period, cancelable after any period:

$$
\text{UpFront}^\triangleright(t_0, x, p, W, D, m) = \\
\text{UpFront}(t_0, x, p, W, D) \triangleright ... \triangleright \\
\text{UpFront}(t_{m-1}, x, p, W, D)
$$

where $t_i = t_0 + ip$ for $i$ from 0 to $m - 1$

**Problems**

• Same question concerning associativity.
• There is still a problem with $t_i$: it is defined but never used!
2.1.3 A License Language

Syntax

The language *DigitalRights* is defined by the following grammar:

\[ e ::= (a \mid u) \] 
\[ (f p \mid [u] m p) \] 
\[ p x (u f | f r | p) \] 
\[ f w o n D \]

Example

until 01/01/03
for up to 12 months
pay $10.00 upfront
for "Jazz Classics"
on "devices registered to license holder"

Semantics

\[ M[z][a t t z] = M_1[z](t) \] 
\[ M[z][u n t l t z] = \bigcup_{t' < t} M_1[z](t') \] 
\[ M_1[z][f p \ z[t_0](p) = M_2[z](t_0, p) \] 
\[ M_1[z][f u p t o m p \ z[t_0](p) = M_2[z](t_0, p) \] 
\[ \] 
\[ M_1[z][f for m p \ z[t_0](p) = M_2[z](t_0, p) \] 
\[ \] 
\[ M_2[z][p x u f p o n D(t, p) = U p F r o n t(t, x, W, D) \] 
\[ M_2[z][p x f r at e f o r W on D(t, p) = F l a t R a t e(t, x, W, D) \] 
\[ M_2[z][p x p e r u s e f o r W on D(t, p) = P e r U s e(t, x, W, D) \]

Problems

- Parameters \((t_0, p)\) in first lefthand side of \(M_1\) is wrong,
- should be just \((t_0)\).

2.1.4 Comments

We have already brought our comments in-line in connection with the formulas.

2.2 An RSL Model

<table>
<thead>
<tr>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>This, as of June 29, 2006, incomplete model is being worked out by the second author.</td>
</tr>
</tbody>
</table>
2.2.1 Actions, Events, Realities and Licences

type
\[ T, W, D, P \]
\[ E = T \times A \]
\[ A == mkR(w:W,d:D) | mkP(x:P) \]
\[ R = \{ | evs:E-set \cdot wfEvs(evs) | \} \]

Annotations

- T, W, D, and P stands for time, work, device and payment.
- Events, e:E, are Cartesian pairs of times (i.e., time stamps) and actions.
- Actions are discrimated, either renderings (which are records mkR(w,d)) of works and devices, or payments (mkP(x)) of currency amounts.
- The trailing “or” shows our schematic way of representing actions.
- Realities, r:R, are well-formed sets of events.

value
\[ wfEvs: E-set \rightarrow \textit{Bool} \]
\[ wfEvs(evs) \equiv \forall (t,a),(t',a'):E \cdot \{(t,a),(t',a')\} \subseteq evs \land t=t' \Rightarrow a=a' \]

\[ r \leq t: \text{prefix}: R \rightarrow T \rightarrow R \]
\[ r \leq t \equiv \text{prefix}(r)(t) \equiv \{(t',a)|(t',a):E \cdot (t',a) \in r \land t' \leq t\} \]

\[ r' \sqsubseteq r: \text{is_prefix}: R \times R \rightarrow \textit{Bool} \]
\[ r' \sqsubseteq r \equiv \exists t:T \cdot \text{is_prefix}(r',r) \equiv \exists t:T \cdot \text{prefix}(r')(t) \]

Annotations

- A set of events form a reality if no two events have the same time stamp.
- The prefix of a reality up to and including time t is the set of all those events of the reality whose time stamp is equal to or less than t.
- A reality is a prefix of another event if there is a time t such that the former reality is a prefix of the latter reality.

type
\[ L = R-set \]

value
\[ /* \text{viable}: capable of working */ \]
\[ is\_viable: R \rightarrow L \rightarrow T \rightarrow \textit{Bool} \]
\[ is\_viable(r')(r)(l)(t) \equiv r \in l \land is\_prefix(prefix(r')(t),r) \]

\[ /* \text{fulfilled}: a consumer reality r' satisfies a license reality */ \]
Specifications of Digital Rights Languages — June 29, 2006

is_fulfilled: \( R \rightarrow L \rightarrow T \rightarrow \text{Bool} \)

is_fulfilled\((r')\)(l)(t) \(\equiv\) prefix\((r')\)(t) \(\in\) l

is_breached: \( R \rightarrow L \rightarrow T \rightarrow \text{Bool} \)

is_breached\((r')\)(l)(t) \(\equiv\) \(\sim\)\(\exists\) \(r : R \mid r \in l \land\) is_viable\((r')\)(r)(l)(t)

Annotations

- A reality \( r' \) is viable at a time \( t \) wrt. a reality \( r \) of a license \( l \) if \( r' \) is a prefix, at that time, of \( r \).
- A reality \( r' \) is, at time \( t \) fulfilled wrt. a license \( l \) if the prefix of \( r' \) at time \( t \) is a reality of the license.
- License \( l \) is breached by consumer reality \( r' \) if there is no reality \( r \) in \( l \) that is viable for \( r' \) at \( t \).

2.2.2 Standard Licences

Syntax

type
\[ I = \text{Std}_L = \text{UpFront} | \text{FlatRate} | \text{PerUse} | \text{Until} | \text{NonCanMuUpFro} | \text{CanMuUpFro} \]

Basics = T \times P \times I \times \text{W-set} \times \text{D-set}

UpFront == mkUF(sb:Basics)
FlatRate == mkFR(sb:Basics)
PerUse == mkPU(sb:Basics)
Until == mkU(sb:Basics,st:T)
NonCanMuUpFro == mkNCMF(sb:Basics,sm:Nat)
CanMuUpFro == mkCMF(sb:Basics,sm:Nat)

Annotations

- \( I \) stands for an interval, i.e., a time period.
- \( \text{Std}_L \) stands for standard licenses.
- There are six forms of standard licences: UpFront, FlatRate, PerUse, Until, NonCanMuUpFro and CanMuUpFro.
- They all share some basic information Basics, a Time limit, Payment amount, Interval, identification of the set of Works covered by the license, and identification of the set of Devises overed by the license.
- The NonCanMuUpFro and CanMuUpFro commands further specify a natural, usually non-zero number (of times of rendering).
- The Std definition defines the set of commands as a union using the | type constructor.
- Each individual type is then defined by distinctly named record type constructors: mkUF, mkFR, mkPU, mkU, mkNCMF and mkCMF.
We have for ease of recalling these mnemonics chosen to name the constructors with an initial \( \text{mk} \) (for ‘make’) and then an abbreviation of the type name being defined.

The \( \ldots \) parts of the body of the record type expressions designate selector functions.

Meta-linguistically:

\[
\begin{align*}
\text{type} & \quad A, B, \ldots, C \\
R &= \text{mkR}(sa:A, sb:B, \ldots, sc:C) \\
\text{axiom} & \\
& \forall r:R, a:A, b:B, \ldots, c:C \cdot \\
& r = \text{mkR}(sa(r), sb(r), \ldots, sc(r)) \land \\
& a = sa(\text{mkR}(a,b,\ldots,c)) \land b = sb(\text{mkR}(a,b,\ldots,c)) \land \ldots \land c = sc(\text{mkR}(a,b,\ldots,c)) \\
\end{align*}
\]

Semantics

\[
M(\text{mkUF}(t_0,x,p,ws,ds)) \equiv \\
\{ \{(t_0,\text{pay}(x))\} \cup \\
\quad \text{let } ls = [ti \mapsto \text{rndr}(w,d)|ti:T, w:W, d:D \cdot ti \in ts \land w \in ws' \land d \in ds'] \text{ in} \\
\quad \{ (ti,ls(ti))|ti:T \cdot ti \in \text{dom} \ ls\} \text{ end} \\
| n:\text{Nat}, ts:T-set, ws':W-set, ds':D-set \cdot \\
\text{card} ts=n \land ws' \subseteq ws \land ds' \subseteq ds \land t_0 < \text{min}(ts) \land \text{max}(ts) \leq t_0+p \}
\]

Annotations

The above formulation follows that of Gunter et al.,

but where their model is plain wrong: it does not designate all combinations of works and devices, ours is right:

\* At \( t_0 \) payment is issued;
\* the above expression\(^1\) has two set comprehensions:
\* \( \{ \{A\} \cup \{B|C\cdot D\} | E\cdot F \} \)
\* The inner comprehension \( \{B|C\cdot D\} \) expresses all possible sequences of \( n \) renderings involving any combination of works \( w \) and devices \( d \) from subsets \( ws \) and \( ds \) of works and devices.
\* The outer comprehension “selects” an arbitrary, finite \( n \), a set \( ts \) of \( n \) time points all of which lies between \( t_0 \) and \( t_0 + p \), and arbitrary subsets \( ws \) and \( ds \) or works and devices of those given,
\* The inner comprehension ensures that all we express all runs of renderings of length \( n \) over all combinations of works and devices.

\(^1\) \{ \{(t_0,\text{pay}(x))\} \cup \{ (ti,\text{rndr}(w,d)) | ti:T, w:W, d:D \cdot ti \in ts \land w \in ws' \land d \in ds' \} | n:\text{Nat}, ts:T-set, ws':W-set, ds':D-set \cdot \text{card} ts=n \land ws' \subseteq ws \land ds' \subseteq ds \land t_0 < \text{min}(ts) \land \text{max}(ts) \leq t_0+p \} \]
The outer comprehension ensures that we express all possible and indefinite length runs.

- A better model would be one which, instead of constructively designating all possible runs, expresses the property that a run is an up front run and all such, for the given arguments, are present.

```plaintext
type
PayEvent = T × ({|pay|} × Nat)
RndEvent = T × ({|render|} × W × D)

UpFrontLicense = {{ℓs:L • wf_UPL(ℓs) }}
```

```plaintext
value
wf_UPL: L → Bool
wf_UPL(ℓ) ≡ ∃ t:T,x: Nat,p:P,ws: W-set,ds: D-set • is_ufl(t0,x,p)(ws,ds,ts)(ℓ)

is_ufl: (T × Nat × P) → (W-set × D-set × T-set) → L → Bool
is_ufl(t0,x,p)(ws,ds,ts)(ℓ) ≡
∃ t': T, x': X • (t',(|pay|,x')) ∈ ℓ ⇒ t'=t0 ∧ x'=x ∧
∀ ti:T, w: W, d: D • ti ∈ ts ∧ w ∈ ws ∧ d ∈ ds ⇒
(t,(render,w,d)) ∈ ℓ ∧
¬∃ (t,(render,w',d')): RndEvent •
(t,(render,w',d')) ∈ ℓ ∧ t' ≠ ts ∧ w' ≠ ws ∧ d' ≠ ds
```

A set ℓs of licenses is the set of all up front licenses designated by M(mkUF(t0,x,p,ws,ds)) if it satisfies are_all_ufl(t0,x,p)(ws,ds,ts)(ℓs).

```plaintext
value
are_all_ufl: T × Nat × P × W-set × D-set → L-set → Bool
are_all_ufl(t0,x,p)(ws,ds,ts)(ℓs) ≡
∀ ℓ:L • ℓ ∈ ℓs ⇒
ws' ⊆ ws ∧ ds' ⊆ ds ∧ card ts=n ⇒ is_ufl(t0,x,p)(ws',ds',ts)(ℓ)
```

**Annotations**

- The “do not exists” clauses shall ensure both largest sets of up front licenses over appropriate time spans, works, and devices and that there is no “junk”.

- Otherwise we leave it to the reader to decipher the formulas.

```plaintext
value
M(mkFR(t,x,p,ws,ds)) ≡
{let ls = [{ti → rndr(w,d)|ti:T, w: W, d: D • ti ∈ ts ∧ w ∈ ws ∧ d ∈ ds'} | ti:T • ti ∈ dom ls} end
∪{{(ti,ls(ti))}|ti:T • ti ∈ dom ls} end
card ts=n ∧ ws' ⊆ ws ∧ ds' ⊆ ds ∧ t0 ≤ min(ts) ∧ max(ts) < tn' ≤ t0 + p}
```
Annotations

- We leave it to the reader to decipher the formulas.

value

\[ M(\text{mkPU}(t,x,p,ws,ds)) \equiv \begin{cases} \text{let } ls = \{ t_i \mapsto \text{rndr}(w,d) \mid t_i \in ts \land w \in ws' \land d \in ds' \} \text{ in} \\ \{(t_i,ls(t_i)) \mid t_i \in \text{dom } ls \} \end{cases} \]

\[ \cup \{ (tn',\text{pay}(n+x)) \mid n:\text{Nat},tn':T,ts:T\text{-set},ws':W\text{-set},ds':D\text{-set} \cdot \text{card } ts=n \land ws' \subseteq ws \land ds' \subseteq ds \land t0 \leq \min(ts) \land \max(ts) < tn' \leq t0+p \} \]

Annotations

- We leave it to the reader to decipher the formulas.

value

\[ M(\text{mkU}(te,x,p,ws,ds)) \equiv \cup \{ M(\text{mkUF}(t_0,x,p,ws,ds)) \mid t_0:T \leq te \} \]

\[ M(\text{mkNCMF}((t,x,p,ws,ds),m)) \equiv \text{UpFront}^\Delta((t,x,p,ws,ds),m) \]

\[ M(\text{mkCMF}((t,x,p,ws,ds),m)) \equiv \text{UpFront}^\triangleright((t,x,p,ws,ds),m) \]

Annotations

- We leave it to the reader to decipher the formulas.

value

\[ \Delta: L^* \rightarrow L \]

\[ \Delta(\ll) \equiv \begin{cases} \text{case } \ll \text{ of} \\ \langle l \rangle \rightarrow l, \\ \langle l \rangle \triangleleft \ll' \rightarrow \{ r \cup r',r',r:R \cdot r \in 1 \land r' \in \Delta(\ll') \} \end{cases} \]

\[ \triangleright: L^* \rightarrow L \]

\[ \triangleright(\ll) \equiv \begin{cases} \text{case } \ll \text{ of} \\ \langle l \rangle \rightarrow l, \\ \langle l \rangle \triangleleft \ll' \rightarrow \{ r,r \cup r',r':R \cdot r \in 1 \land r \neq \{} \land r' \in \triangleright(\ll') \} \end{cases} \]

value

\[ \text{UpFront}^\Delta: \text{Basics} \times \text{Nat} \rightarrow L \]

\[ \text{UpFront}^\Delta((t,x,p,ws,ds),m) \equiv \Delta((M(\text{mkUF}(ti,x,p,ws,ds))|i:\text{Nat},ti:T\cdot i:0..m-1 \land ti=t0+i\times p)) \]

\[ \text{UpFront}^\triangleright: \text{Basics} \times \text{Nat} \rightarrow L \]

\[ \text{UpFront}^\triangleright((t,x,p,ws,ds),m) \equiv \triangleright((M(\text{mkUF}(ti,x,p,ws,ds))|i:\text{Nat},ti:T\cdot i:0..m-1 \land ti=t0+i\times p)) \]
2.2.3 A License Language

type
DRExpr = Time × Repetition × Payment × WorksDevices
Time == mkAt(t:T) | mkUntil(t:T)
Repetition == mkFor(p:I) | mkRepeat(m: Nat, p:I) | mkUpTo(m: Nat, p:I)
Payment = P × Kind
Kind == upfront | flatrate | peruse
WorksDevices = W-set × D-set

Annotations

•
•
•

value
M0(mkAt(t),r,(x,k),wds) ≡ M1(r,(x,k),wds)(t)
M0(mkUntil(t),r,(x,k),wds) ≡ ∪{M1(r,(x,k),wds)(t′)|t′:T•t′<t}
M1(mkFor(p),(x,k),wds)(t) ≡ M2(x,k)(t,p)
M1(mkRepeat(m,p),(x,k),wds)(t) ≡
▷{(M2((x,k),wds)(ti,p)|i: Nat, ti:T•i: {0..m−1}∧ti=t0+i×p)}
M1(mkUpTo(m,p),(x,k),wds)(t) ≡
Δ{(M2((x,k),wds)(ti,p)|i: Nat, ti:T•i: {0..m−1}∧ti=t0+i×p)}
M2((x,upfront),(ws,ds))(t,p) ≡ M(mkUF(t,x,p,ws,ds))
M2((x,upfront),(ws,ds))(t,p) ≡ M(mkFR(t,x,p,ws,ds))
M2((x,upfront),(ws,ds))(t,p) ≡ M(mkPU(t,x,p,ws,ds))

2.3 A CafeOBJ Model

The CafeOBJ model was first proposed by the first author. It has then been (is being) worked on by the first and the last two authors.

2.3.1 CafeOBJ Specification

mod* LIST (X :: TRIV) {
pr (EQL) -- a new built-in module for the equality (_,_) predicate

[ Nil NnList < List ] -- Nil Non-nil-List < List

op nil : -> Nil
op cons : Elt List -> NnList

op hd : NnList -> Elt -- head of the list
op tl : List -> List -- tail of the list

vars E E' : Elt
var L : List
var NnL : NnList

eq hd(cons(E, L)) = E .
eq tl(nil) = nil .
eq tl(cons(E, L)) = L .

-- define inequality
eq (nil = NnL) = false .
}

-- enhanced List for license
mod* LIST+ {
  pr ( LIST * { op cons -> _;_; } )
    -- rename operator cons to _;_ for convenience
  pr ( NAT ) -- NAT is a built-in system module in CafeOBJ

  op _in_ : Elt List -> Bool -- check whether an element is in a list
  op r-hd_ : NnList -> Elt -- reverse-head
  op r-tl_ : List -> List -- reverse-tail
  op length_ : List -> Nat -- count the number of elements of a list

  vars E E' : Elt
  vars L L' : List
  var NnL : NnList

  eq E in (E' ; L) = (E = E') or (E in L) .
eq E in nil = false .
eq r-hd(E ; NnL) = r-hd(NnL) .
eq r-hd(E ; nil) = E .
eq r-tl(E ; L) = if L = nil then nil else (E ; r-tl(L)) fi .
eq r-tl(nil) = nil .
eq length(E ; L) = 1 + length(L) .
eq length(nil) = 0 .
-- -- test LIST+
open LIST+
ops E1 E2 E3 : -> Elt .
red r-hd(E1 ; E2 ; E3 ; nil ) .
red hd(E1 ; E2 ; nil) .
red hd(tl(E1 ; E2 ; nil)) .
close
-- -- end of test LIST+
-- eof

mod* PERIOD {
  [Period]
}

mod* TIME {
  pr(PERIOD + EQL)
  [Time]
  op _+_ : Time Period -> Time
  op _<=_ : Time Time -> Bool
  vars T T' T'' : Time
  var P : Period
  eq T <= (T + P) = true .
  -- totally ordered by <=
  -- reflectivity
  eq T <= T = true .
  -- anti-symmetry
  eq T <= T and T <= T = true .
  -- transitivity
  ceq T <= T' = true
    if T <= T' and T' <= T' .
  -- comparability
  eq (T <= T' or T' <= T) = true .
  -- inequality between time
  ceq (T = T') = false if ((T <= T') implies not (T' <= T)) .
}

mod* WORK {
  [ Work ]
}
mod* DEVICE {
    [ Device ]
}

mod* ACTION {
    pr ( EQL )
    pr ( WORK + DEVICE + NAT + STRING )

    [ Action ]

    op render : Work Device -> Action
    op pay : Nat -> Action

    op n-act : Action -> String -- get the name of an action
    op w-act : Action -> Work -- get the work of an action
    op d-act : Action -> Device -- get the device of an action

    var W : Work
    var D : Device
    vars X X' X'' : Nat -- payment

    eq w-act(render(W, D)) = W .
    eq d-act(render(W, D)) = D .

    eq n-act(pay(X)) = "pay" .
    eq n-act(render(W, D)) = "render" .

    -- inequality between pay and render actions
    eq ("pay" = "render") = false .

    -- inequality X and X'
    ceq (X = X') = false if (X < X' or X' < X) .
    ceq (X * X' = X * X'') = false if not (X' = X'') .

    -- inequality pay(X) and pay(X')
    ceq (pay(X) = pay(X')) = false if not (X = X') .
}

mod* EVENT {
    pr ( 2TUPLE ( C1 <= view to TIME { sort Elt -> Time } ,
               C2 <= view to ACTION { sort Elt -> Action } ) )

    * { sort 2Tuple -> Event,
       op 1* _ -> t _ ,
       op 2* _ -> a _ ,
       op << _. ; _ >> -> _ & _ )
}
mod* REALITY {
-- instantiation of parameterized module LIST+, i.e., reality is a list
-- of events, and rename sort List to sort Reality
pr ( LIST+ ( X <= view to EVENT { sort Elt -> Event } )
   * { sort List -> Reality ,
       sort Nil -> NilReality,
       sort NnList -> NnReality,
       op nil -> nilR } )

-- define the corresponding time list of the reality
pr ( LIST+ ( X <= view to TIME { sort Elt -> Time } )
   * { sort List -> TimeList ,
       sort Nil -> NilTL,
       sort NnList -> NnTL,
       op nil -> nilTL } )

op errR : -> ?Reality
op _<=_ : Reality Time -> Reality -- prefix of reality
op _<=_ : Reality Reality -> Bool -- R is a prefix of R’?

vars T T’ : Time
vars A A’ : Action
vars E E’ : Event
vars R R’ : Reality

op times : Reality -> TimeList
-- derive the corresponding time list of reality
eq times (E ; R) = (t(E) ; times(R)) .
eq times(nilR) = nilTL .

-- get a sorted Reality list
ceq (E ; E’ ; R) = (E’ ; E ; R) if not (t(E) <= t(E’)) .

-- events occur at distinct times
-- ceq (E ; E’ ; R) = (E’ ; R) if t(E) = t(E’) .
    ceq (E ; E’ ; R) = errR if t(E) = t(E’) .
eq (E ; errR) = errR .

-- A list which allows events to occur at the same time as of
-- other events in the list is no longer a Reality.
-- ceq (E ; R) = errR if (t(E) in times(R)).

-- define the axioms for <=. Notice <= is also defined in TIME
ceq ((E ; R) <= T) = (E ; (R <= T)) if t(E) <= T.
ceq ((E ; R) <= T) = nilR if not (t(E) <= T).
eq (nilR <= T) = nilR.

-- axioms checking whether R <= R'
-- Here a trick is that if R is a prefix of R', then R must be a
-- sublist of R'.
eq (E ; R <= E' ; R') = if E = E' then (R <= R') else false fi.
eq (R <= nilR) = false.
eq (nilR <= R) = true.

-- test
--
-- open REALITY
--
-- ops e e' : -> Event.
-- ops r r' : -> Reality.
-- ops t t1 t2 : -> Time.
-- ops a0 a1 a2 : -> Action.
--
-- red (3 & a0 ; 1 & a1 ; 2 & a2 ; nilR).
-- red (2 & a0 ; 2 & a0 ; 4 & a2 ; 2 & a2 ; nilR).
-- red (3 & a0 ; nilR).
--
-- red (1 & a1 ; 2 & a2 ; nilR) <= 3.
-- red (1 & a1 ; 2 & a2 ; nilR) <= 1.
--
-- close
--
-- eof
--

-- define module WORKLIST, i.e., the list of works.
mod* WORKLIST {
  pr ( LIST+ ( X <= view to WORK { sort Elt -> Work } )
   * { sort List -> WorkList ,
      sort Nil -> NilWL,
      sort NnList -> NnWL,
      op nil -> nilWL } )
}


-- define module DEVICELIST, i.e., the list of devices
mod* DEVICELIST {
  pr ( LIST+ ( X <= view to DEVICE { sort Elt -> Device } )
    * { sort List -> DeviceList ,
        sort Nil -> NilDL,
        sort NnList -> NnDL,
        op nil -> nilDL } )
}

-- License is a list of Reality, which is also a list of Event.
mod* LICENSE {
  pr ( LIST+ ( X <= view to REALITY { sort Elt -> Reality } )
    * { sort List -> License ,
        sort Nil -> NilLic,
        sort NnList -> NnLic
        op nil -> nilL } )
  pr ( WORKLIST + DEVICELIST )
  op cr : -> Reality -- cr is the Complete Reality
  op _|_ : Reality License -> Reality
  -- a Reality of a License is viable for a Reality given by
  -- associating the complete reality with the License at a time
  op viable : Reality License Time -> Bool
  -- a License is fulfilled by a Reality given by associating
  -- the complete reality with the License at a time
  op fulfilled : License Time -> Bool
  -- a License is breached by a Reality given by associating
  -- the complete reality with the License at a time
  op breached : License Time -> Bool
  -- Check whether all the actions in a reality are render actions,
  -- and the works and devices of the actions are in the WorkList
  -- and DeviceList, respectively.
  op all-render : Reality WorkList DeviceList -> Bool
  -- Instead of defining UpFront, FlatRate, and PerUse as sorts,
  -- we defined them as predicates which checks if a Reality belongs
  -- to each standard licenses.
--- The sets of Realities which make the predicates held are standard
--- licenses.

--- Check if a Reality is the one which should occur in UpFront License
op isUpFrontR : Reality Time Nat Period WorkList DeviceList -> Bool

--- Check if a Reality is the one which should occur in FlatRate License
op isFlatRateR : Reality Time Nat Period WorkList DeviceList -> Bool

--- Check if a Reality is the one which should occur in PerUse License
op isPerUseR : Reality Time Nat Period WorkList DeviceList -> Bool

vars R Rl : Reality
var L : License
vars T T0 T1 Tn : Time
var P : Period -- time period
var X : Nat -- amount of payment
var W : Work
var D : Device
var WS : WorkList
var DS : DeviceList
vars E E’ : Event
vars R R’ : Reality
var NnR : NnReality -- Non-nil-Reality

-- axioms defining viable
eq viable(R, L, T) = (((cr | L) <= T) <= R) and (R in L) .

-- axioms defining fulfilled
eq fulfilled(L, T) = (((cr | L) <= T) in L) .

-- axioms defining breached
eq breached((R ; L), T) = breached(L, T)
and not (((cr | L) <= T) <= R) .

eq breached(nilL, T) = true .

-- axioms defining all-render
eq all-render(nilR, WS, DS) = true .
eq all-render((E ; R), WS, DS) = ((n-act(a(E))) = "render") and
(w-act(a(E)) in WS) and
((d-act(a(E))) in DS) and
all-render(R, WS, DS) .

-- axioms defining isUpFrontR
eq isUpFrontR(nilR, T0, X, P, WS, DS) = false .
eq isUpFrontR((E ; nilR), T0, X, P, WS, DS) = (E = pay(X)) .
eq isUpFrontR((E ; NnR), T0, X, P, WS, DS) = ((n-act(a(E)) = "pay") and
(T0 <= t(E)) and all-render(NnR, WS, DS) and (t(r-hd(NnR)) <= T0 + P)) .

-- axioms defining isFlatRateR
eq isFlatRateR(nilR, T0, X, P, WS, DS) = false .

eq isFlatRateR((E ; R), T0, X, P, WS, DS) = (T0 <= t(E) and all-render(r-tl(E ; R), WS, DS) and a(r-hd(E ; R)) = pay(X) and t(r-hd(E ; R)) <= T0 + P) .

-- axiom defining isPerUseR
eq isPerUseR(nilR, T0, X, P, WS, DS) = false .

eq isPerUseR((E ; R), T0, X, P, WS, DS) = T0 <= t(E) and all-render(r-tl(E ; R), WS, DS) and a(r-hd(E ; R)) = pay(length(r-tl(E ; R)) * X) and t(r-hd(E ; R)) <= T0 + P .

2.3.2 Discussion

The above CafeOBJ specification models three standard licenses, i.e., Flat Rate, Up Front, and Per Use, based on initial algebra and specification, no hidden algebra or behavioral specification (such as observational transition systems, OTSs) are introduced. The three authors tried to model the licenses from scratch, first defining basic module of LIST and then specifying the EVENT, REALITY, and LICENSE step by step. The intention is not only to derive an alternative formal specification written in CafeOBJ based on Gunter’s model and definition, but more importantly, to make the specification executable as for the subsequent formal verification. Some extra and fundamental axioms (equations) have been added to achieve this goal which has not been discussed in Gunter’s paper and in the RAISE specification. The authors discussed several times and exchanged ideas before finishing the specification, and this is why the main parts of the three specifications (licenses) looks similar. However, difference still existed, and bugs and errors no doubt also have not been kicked out completely, further improvement and revising is needed.

Through the practice, two important techniques of CafeOBJ have been studied as follows, which are the foundation for future study.

- Parameterized modules are used to construct List, Reality (a list of Event), and License (which is a list of Reality, i.e., a list of list)
- Behavioral Specification (such as behavioral List) has also been practiced though finally it is not represented in this report.
2.4 Comparisons
Chapter 3

The Pucella/Weissman Paper

3.1 A Transcript of the Pucella/Weissman Paper

This transcript was kindly provided by the third author. It has been slightly edited by the fourth author. More editing work will be done.

3.1.1 The Logic $L^{lic}$

They introduce a logic, $L^{lic}$. Formulas in $L^{lic}$ include permission and obligation operators, as well as temporal operators.

Syntax

The syntax of Logic $L^{lic}$

- formulas($\varphi, \psi, ...$),
- actions($\alpha, ...$),
- licenses($\ell, ...$).

Their definitions assume a set $Names$ of License names, a set $works$ of works, and a set $Devices$ of devices. Actions are taken from a set $Act = \{\text{render}[w, d] : w \in Works, d \in Devices\} \cup \{\text{pay}[x] : x \in \mathbb{R}\} \cup \{\bot\}$, where $\bot$ represents the null or "do nothing" action. Also, the let $Lic$ be the set of licenses $\ell$.

Here is the formal description:

- $\varphi ::= n : \ell | \alpha | P \alpha | \varphi_1 \land \varphi_2 | \neg \varphi | \lozenge \varphi | \varphi_1 \cup \varphi_2$
- $\alpha ::= (a, n)$
- $\ell ::= a|\ell_1 \ell_2 | \ell^* | \ell_1 \cup \ell_2$

Where $n \in Names$ and $a \in Act$.
Example
An owner of an online journal requires a fee to be paid before each access.

• The license $\ell$ can be written in the logic as:

$$\ell = ((\text{pay}[\text{fee}](\bot)^* \text{render}[\text{journal}, d]) \cup \bot)^*$$

where $d$ is the device that the client uses to access the journal.

• The license is labeled $n$, the property that the client is not obligated to access the journal immediately after paying the fee can be written as:

$$(\text{pay}\ [\text{fee}] \Rightarrow \circ (\neg \text{render}_n[\text{journal}, d]))$$

• The specification that the client does not violate the license can be written as the family of formulas:

$$n : \ell \Rightarrow (\alpha \Rightarrow (P\alpha) \land ((O\alpha) \Rightarrow \alpha))$$

where $\alpha \in \{\text{pay}_n[\text{fee}], \text{render}_n[\text{journal}, d], \bot\}$.

• During one time period, the client pays $1500 on the mortgage $m$, but does not pay the journal $n$ fee as:

$$\text{pay}_m[1500] \land \overline{\text{pay}_n[\text{fee}]}.$$ 

Semantic
The semantics base on the notion of a run
A run is a function:

$$r : \mathbb{N}^- \rightarrow \wp(\text{Names} \times \text{Lic}) \times \text{Act}^\text{Names}$$

They impose the restriction that, at most, one action per time per named license can occur.

Finally, a license $(n : \ell)$ is active at time $t$ in run $r$ if there exists a time $t' \leq t$ such that $(n : \ell) \in \text{lic}(r, t')$

While a run captures the client’s actions, an interpretation states what is permitted. Formally, a permission interpretation $P$ is a function

$$P : \mathbb{N}^- \rightarrow \wp(\text{Act} \times \text{Names})$$

Intuitively, if $(a, n) \in P(t)$ then at time $t$, the client is permitted to perform action $a$ with respect to license name $n$.

The definition of trace

• A trace refers to a sequence of actions.
• The notation \( s_1 \cdot s_2 \) denotes the concatenation of two sequences of actions \( s_1 \) and \( s_2 \) where \( s_1 \cdot s_2 = s_1 \) if \( s_1 \) is infinite.

• A trace \( s_1 \) is said to be a prefix of trace \( s_2 \) if there is some trace \( s \) such that \( s \cdot s = s_2 \)

Function \( L[\ell] \) and \( I[\ell] \)

Definition of \( L[\ell] \)

Induction on the structure of a given license \( \ell \):

\[
L[\ell] = \{ a \}
\]

\[
L[\ell_1 \ell_2] = \{ s_1 \cdot s_2 : s_1 \in L[\ell_1] \text{and} s_2 \in L[\ell_2] \}
\]

\[
L[\ell_1 \cup \ell_2] = L[\ell_1] \cup L[\ell_2]
\]

\[
L[\ell^*] = \bigcup_{n \geq 0} \{ s_1 \cdot \ldots \cdot s_n : s_i \in L[\ell] \}
\]

Definition of \( I[\ell] \)

\[
I[\ell] = \{ s \cdot \bot^\infty : s \in L[\ell] \}
\]

Then, they define the interpretation \( P_r \) corresponding to run \( r \). Given a named license \((n, \ell)\) issued at time \( t_1 \) in a run \( r \), the action-sequence of \( n \) up to time \( t_2 \), denoted \( r[n, t_2] \), is the sequence \( a_0 a_1 \ldots a_{t_2-t_1-1} \) such that:

\[
a_i = \begin{cases} a & \text{if} (a, n) \in \text{act}(r, t_1 + i) \\ \bot & \text{otherwise} \end{cases}
\]

The interpretation \( P_r \) is, for all times \( t \geq 0 \), \( P_r(t) \) is the smallest set such that for all license names \( n \in Names \) and actions \( a \in Act, (\bot, n) \in P(t) \) if the license \((n, \ell)\) is not active and \((a, n) \in P(t) \) if the license is active and \( r[n, t] \cdot a \) is viable for \( \ell \).

In order to associate the action expression \( \alpha \) with name-action pairs. They defined a mapping \( A[\alpha] \) from expressions to sets of pairs.

\[
A[(a, n)] = \{(a, n)\}
\]

\[
A[[a, n]] = \{(b, n) | b \neq a\}
\]

Then define what it means for a formula \( \varphi \) to be true (or satisfied) at a run \( r \) at time \( t \), written \( r, t \models \varphi \)

• \( r, t \models n : \ell \text{ if } (n, \ell) \in \text{lic}(r, t) \),

• \( r, t \models \alpha \text{ if } \exists (a, n) \in A[\alpha] \text{ s.t. } (a, n) \in \text{act}(r, t) \),

• \( r, t \models P \alpha \text{ if } \exists (a, n) \in A[\alpha] \text{ s.t. } (a, n) \in P_r(t) \),
• \( r, t \models o\varphi \) if \( r, t + 1 \models \varphi \)
• \( r, t \models \text{if} \) for all \( t' \geq t, r, t' \models \varphi \),
• \( r, t \models \varphi U \psi \) if \( \exists t' \geq t, s.t. r, t' \models \psi \) and \( r, t'' \models \varphi \) for all \( t'' > t' \geq t \),
• \( r, t \models \neg\varphi \) if \( r, t \not\models \varphi \),
• \( r, t \models \varphi \land \psi \) if \( r, t \models \varphi \) and \( r, t \models \psi \).

• Various properties of permission (\( P \)) and obligation (\( \neg P(\bar{a}, n) \)) follow from the above semantics.
• An action is obligated if and only if it is the only permitted action.
For all action expressions \((a, n)\), the formula \( P(a, n) \lor P(\bar{a}, n) \) is valid. Hence,
• If \((P(\bar{a}, n))\) is not true at a point, \( P(a, n) \) must be true.
• \( O(a, n) \Rightarrow P(a, n) \) is valid
In particular
• The validity of \( O(a, n) \Rightarrow P(a, n) \) indicates that obligation should be read as "must" and not as "ought"
• We can not express conflicting prohibitions and obligations in this framework

3.1.2 Encoding finite runs and licenses
3.1.3 Handling different license languages
3.1.4 Comments
We have already brought our comments in-line in connection with the formulas.

3.2 A RAISE/RSL Specification
3.2.1 Three Sublanguages
There are three languages involved in this language of licences:
• an action command and an event command language, A and E,
• a license language, L, and
• a modal propositional reasoning language, \( \Phi \).

type
\( A, E, L, \Phi \)

Actions are performed by consumers. Performing an action constitutes an event. Events name licenses. Named licences are issued by owners. They define which sequences of actions a consumer who subscribes to the named license can perform, which not. The modal propositional language is defined for use by consumers and owners.
3.2.2 The Action Command and Event Command Languages

The Action Language

There are basically three actions: payment (\texttt{pay}\,[x] where \(x\) is some amount of monies, \(x:M\)), rendering (\texttt{render}\,[w,d] where \(w\) names some licensed work, \(w:W\), and \(d\) names some device that can render the work, \(d:D\)) and in-action (\texttt{skip}).

\begin{verbatim}
    type
    M, W, D
    A == mkPay(x:M) | mkRnd(w:W,d:D) | skip
\end{verbatim}

Annotations
- \(M, W,\) and \(D\) designator money amounts, works and devices.
- \(A\) are action commands.
- \texttt{skip} designates the no action command.
- We explain the ==, \texttt{mkPay} and \texttt{mkRnd} symbols in Sect. 3.2.4.

The Event Language

An event is the occurrence of a license named action.

\begin{verbatim}
    type
    N
    E = N \times A /* concretely: (n,a) */
\end{verbatim}

Annotations
- \(N\) designate license names.
- \(E\) designate events
- and defines them as pairs (Cartesians) of license names and action commands.

Event Complements

The idea of an event complement is that of expressing, in propositions, (either explicitly the actions that may be performed or) implicitly the complement of actions (with respect to an explicitly stated action) that may be performed:

\begin{verbatim}
    type
    CE == mkCE(n:N,a:A) /* concretely: (n,\overline{a}) */
\end{verbatim}

Annotations
- \(CE\) designates complement action commands
- and defines them as constructed records (i.e., named Cartesians).
- We explain the == and \texttt{mkCE} symbols in Sect. 3.2.4.
3.2.3 The License Language

The license language is given by a regular expression, $L$. The regular expression is defined recursively:

\[
L == \text{mkAL}(a:A) | \text{mkCL}(L_1,L_2) | \text{mkLC}(L) | \text{mkUL}(L_1,L_2)
\]

**Annotations**

- $L$ designates license expressions and defines them as the discriminated union ($\mid$) of either
  - action commands, $\text{mkAL}(a)$, or
  - concatenation of license expressions, $\text{mkCL}(L_1,L_2)$, or
  - the closure, $^*$, of a license expression, $\text{mkLC}(L)$ or
  - the union of license expressions, $\text{mkCL}(L_1,L_2)$,
- where $a$ are action commands and $L_1$, $L_2$ and $L$ are license expressions.
- We explain the $==$ and $\text{mkAL}$, $\text{mkCL}$, $\text{mkLC}$ and $\text{mkUL}$ symbols in Sect. 3.2.4.

3.2.4 The Modal Language of Propositions

The language $\Phi$ of propositions includes the modal operators

- $P$: permission. $P\alpha$: action expression $\alpha$ is permitted, i.e., is allowed.
- $O$: obligation. $O\alpha$: action expression $\alpha$ is obligated, i.e., must be honoured (performed).
- $\circlearrowleft$: next. $\circlearrowleft\phi$: $\phi$ holds next.
- $\square$: always. $\square\phi$: $\phi$ always holds, from now on.
- $\diamondsuit$: sometimes. $\diamondsuit\phi$: $\phi$ holds sometime (in the future).
- $;$: chop. $\phi;\psi$: first $\phi$ holds, then $\psi$ holds.

Obligation can be defined in terms of permission.

$$O(a,n) \equiv P(a,n)$$

The sometimes modality can be defined in terms of negation and the always modality:

$$\diamondsuit \phi \equiv \neg \square \neg \phi$$

Now we can present the language $\Phi$: 
type

\[ \Phi = \text{NL} | \text{NA} | \text{NCA} | \text{PA} | \text{OA} | \text{And} | \text{Or} | \text{Neg} | \text{Nxt} | \text{Alw} | \text{Som} | \text{Chp} \]

\[ \text{NL} \equiv \text{mkNL}(\text{sn:}N, \text{sl:L}) \]

\[ \text{NA} = \text{NDA} | \text{NCA} \]

\[ \text{NDA} \equiv \text{mkNDA}(\text{sn:}N, \text{sa:A}) \]

\[ \text{NCA} \equiv \text{mkNCA}(\text{sn:}N, \text{sa:A}) \]

\[ \text{PA} \equiv \text{mkPA}(\text{sna:}\text{NA}) \]

\[ \text{OA} \equiv \text{mkOA}(\text{sna:}\text{NA}) \]

\[ \text{And} \equiv \text{mkAnd}(\text{sl:\}\Phi, \text{sr:\}\Phi) \]

\[ \text{Or} \equiv \text{mkOr}(\text{sl:\}\Phi, \text{sr:\}\Phi) \]

\[ \text{Neg} \equiv \text{mkNeg}(\text{s:\}\Phi) \]

\[ \text{Nxt} \equiv \text{mkNxt}(\text{s:\}\Phi) \]

\[ \text{Alw} \equiv \text{mkAlw}(\text{s:\}\Phi) \]

\[ \text{Som} \equiv \text{mkSom}(\text{s:\}\Phi)m \]

\[ \text{Chp} \equiv \text{mkChp}(\text{sl:\}\Phi, \text{sr:\}\Phi) \]

Annotations

- The language of modal, propositional expressions, \( \Phi \), is here defined as the discriminated union of 12 different types of expressions: \( \text{NL}, \text{NA}, \text{NCA}, \ldots, \text{Saom}, \text{CHP} \).
- The \( \Phi \) definitions just define them as a union using the \( | \) type constructor.
- Each individual type is then defined by distinctly named record type constructors: \( \text{mkNL}, \text{mkNDA}, \text{mkNCA}, \ldots, \text{mkSom}, \text{mkChp} \).
- We have for ease of recalling these mnemonics chosen to name the constructors with an initial \( \text{mk} \) (for ‘make’) and then the same typer name as being defined.
- The \( \text{s...:} \) parts of the body of the record type expressions designate selector functions.
- Meta-linguistically:

\[
\text{type} \\
\text{A, B, \ldots, C} \\
\text{R} \equiv \text{mkR}(\text{sa:A, sb:B, \ldots, sc:C})
\]

\[
\text{axiom} \\
\forall r:R, a:A, b:B, \ldots, c:C \cdot \\
r = \text{mkR}(\text{sa}(r), \text{sb}(r), \ldots, \text{sc}(r)) \land \\
a = \text{sa}(\text{mkR}(a,b,\ldots,c)) \land b = \text{sb}(\text{mkR}(a,b,\ldots,c)) \land \ldots \land c = \text{sc}(\text{mkR}(a,b,\ldots,c))
\]

- We give some “pseudo abstract” forms of the grammar above:

\[
\phi = n:l \\
| \alpha \\
| \phi_l \land \phi_r \\
| \phi_l \lor \phi_r \\
| \sim \phi \\
| \text{O} \phi \\
| \text{O} \phi_l ; \phi_r
\]
3.2.5 The “Meaning” of Licences

Recall the syntax of licenses. The meaning of a license is now taken as the set of traces denoted by a license. The only atomic elements of licenses are action commands. So the meaning of a license is going to be defined as a set of possibly infinite sequences of action commands, namely those permitted by the license, that is, allowed to be performed by licensees, i.e., subscribers to the license.

**type**

\[
\text{AL} = A^* \\
\text{LS} = \text{AL-infset}
\]

**value**

\[
\text{M: L} \to \text{LS} \\
\text{M(l)} \equiv \begin{align*}
\text{case } l\text{ of} \\
\text{mkAL}(a) & \to \{\langle a \rangle\}, \\
\text{mkCL}(l_1,l_2) & \to \{l_1 \uplus l_2|l_1: \text{AL} \land l_2 \in \text{M(l_2)}\}, \\
\text{mkLC}(l') & \to \{\text{conc}((l_j|l_j: \text{AL} \land l_j \in \text{M(l')} \land i \in \langle 1..n \rangle)|n: \text{Nat}\}, \\
\text{mkUL}(l_a,r_a) & \to \{l|l: \text{AL} \land l \in \text{M(l_a)} \cup \text{M(r_a)}\}
\end{align*}
\]

**conc:** \text{AL} \to \text{AL}

\[
\text{conc}(lal) \equiv \begin{align*}
\text{case } lal\text{ of} \\
\langle \rangle & \to \langle \rangle, \\
\langle al \rangle \uplus lal' & \to al \uplus \text{conc}(lal')
\end{align*}
\]

**Annotations**

- AL names the type of finite lists of A elements.
- Recall that \text{skip} is in A.
- LS names the type of finite and infinite sets of AL elements.
- M applies to licenses l:L and yields values of type LS.
- An list expression textsfa (in A) denotes the singleton set of the singleton list whose only element is a.
- A concatenated list expression ll \uplus lr denotes the set of concatenated lists l1 and l2 where l1 is in M(l1) and l2 is in M(l2).
- A closure list expression l^* denotes the infinite set of lists which are the concatenations of any number (n) of lists lj drawn from M(l'), each such contributing list (the ith) has i bound in the inner, the list comprehension as ranging from 1 to n, where n is bound in the outer, the set comprehension, to any natural number.
• A union list expression $l \cup ra$ denotes the set of lists $l$ drawn from either $M(la)$ or $M(ra)$.

We extend the finite traces of action commands by appending to their tail infinite sequences of the `skip` (i.e., do nothing) action command.

**type**

\[
\begin{align*}
FSkipL &= \{\text{\texttt{skip}}\}\ast \\
ISkipL' &= \{\text{\texttt{skip}}\}\omega \\
ISkipL &= \{isl:\text{ISkipL'} \cdot isl \not\in FSKil\}
\end{align*}
\]

**value**

\[
\text{Annotations}
\]

- $FSkipL$ names the type of finite lists of the `skip` no action command.
- $ISkipL'$ names the (sub)type of finite and infinite lists of the `skip` no action command.
- $ISkipL$ names the (sub)type of only infinite lists of the `skip` no action command.
- Infiniteness is achieved by restricting lists $isl$ from not belong to $FSkipL$.
- Of course, $ISkipL$ consists of just one value: the infinite list of `skip` literals.

We finally extend the meaning of license expressions by defining $M$ which when applied $M(l)$ to a license expression $l$ yields traces of actions, $\ell$ as defined by $M(l)$ extended “into infinity” by concatenating $\ell$ with infinite traces of `skips`:

**type**

\[
IAL = \{| al\{isl \cdot al:\text{AL},isl:\text{ISkipL} \}}
\]

**value**

\[
M: L \to IAL \\
M(l) \equiv \{al\{isl|al:\text{AL},isl:\text{ISkipL} \cdot al \in M(l)\}
\]

**Annotations**

- $IAL$ names the type whose values are infinite lists of action commands
- such that a finite prefix is a list of any action commands, including `skip`
- and the rest is the infinite list of `skip` commands.

We define a function, `viable`, which when applied to a sequence, $al$, of action commands and a license expression, $l$, yields truth if $al$ is a prefix of some trace in $M(l)$:

**value**

\[
viable: \text{AL} \times L \to \text{Bool} \\
viable(al,l) \equiv \exists ll:\text{IAL} \cdot ll \in M(l) \land \exists n:\text{Nat} \cdot al = (ll(i)\{i:\text{Nat} \cdot 1 \leq i \leq n\})
\]
3.2.6 The Two “Player” License/Action “Game”

There are two sets of “players” at work. There are the license owners, i.e., the producers or owners of works, \(w:W\), and there are the licensees, i.e., the consumers who own devises, on which to render the works.

At any one point in a discrete space of time one or more (but a finite set of) producers may insert a named license, i.e., offer it for use, and one or more (but a finite set of) consumers may issue distinctly license-named commands. We call such aggregation of issued licenses and events a configuration.

Configurations

A configuration is a pair of sets of distinctly named licences and sets of distinctly named events.

type

\[\Theta' = (N \times L)\text{-set} \times (N \times A)\text{-set}\]

\[\Theta = \{\theta:\Theta' \cdot \text{wf}\Theta(\theta)\}\]

value

\[\text{wf}\Theta: \Theta \to \text{Bool}\]

\[\text{wf}\Theta(nls,nas) \equiv \forall (n,l),(n',l'):(N\times L) \cdot \{(n,l),(n',l')\} \subseteq nls \Rightarrow n=n' \Rightarrow l=l' \wedge \]

\[\forall (n,a),(n',a'):(N\times A) \cdot \{(n,a),(n',a')\} \subseteq nas \Rightarrow n=n' \Rightarrow a=a'\]

Annotations

• A configuration \(\theta:\Theta\) is a pair of sets of

• name license pairs and name action command pairs

• such that there are no two otherwise distinct pairs in respective sets with the same license name,

Runs

So the “game played” starts at some time, say time 0. And at any one late point there has evolved a game which we model as a run. A run is a sequence of configurations.

type

\[T = \text{Nat}\]

\[\text{RUN'} = T \rightarrow \Theta\]

\[\text{RUN} = \{\text{run:RUN'} \cdot \text{wfRUN}(\text{run})\}\]

value

\[\text{wfRUN}: \text{RUN'} \to \text{Bool}\]

\[\text{wfRUN}(\text{run}) \equiv 0 \in \text{dom run} \land \forall t:T \cdot 0 \leq t \leq \text{max dom run} \Rightarrow t \in \text{dom run}\]

\[\text{max: Nat-set} \sim \text{Nat}\]

\[\text{max}(\text{ns}) \equiv \text{let } m: \text{Nst} \cdot m \in \text{ns} \land \exists m': m' \in \text{ns} \land m' > m \text{ in } m \text{ end}\]

\[\text{pre ns} \neq \{\}\]
Annotations

• Time is discrete and is here modelled as natural numbers.
• Any run is a map from times to configurations
• such that all times between 0-time and the maximum time of the mapping
• are in the run.

License and Action Observers on Runs

We define two auxiliary functions:

\[
\text{value}
\]

\[
\text{lics}: \text{RUN} \rightarrow T \rightarrow (N \times L)\text{-set}
\]

\[
\text{acts}: \text{RUN} \rightarrow T \rightarrow (N \times A)\text{-set}
\]

\[
\text{lics}(\text{run})(t) \equiv \text{if } t \in \text{dom run then let } (ls,\_)=\text{run}(t) \text{ in ls else chaos end end}
\]

\[
\text{acts}(\text{run})(t) \equiv \text{if } t \in \text{dom run then let } (\_,as)=\text{run}(t) \text{ in as else chaos end end}
\]

Annotations

• \(\text{lics}(\text{run})(t)\) produces a set of pairs of (unique) license names and licenses.
• \(\text{acts}(\text{run})(t)\) produces a set pairs of (unique) license names and action commands.
• If the time argument provided “falls after” the maximum time of the run then the undefined result is yielded.
• Hence the functions are partial.

Specifically Licensed Actions over Runs

Given a run, \(\text{run}\), and given a license name, \(n\), of a license defined at time \(t1\) in run \(\text{run}\) we seek the sequence of action commands, \(\text{actions}(\text{run},n,t2)\), issued in the run for every time \(ti\) as from time \(t1\) up to time \(t2\). If no action relative to the license named \(n\) was performed at time \(ti\) we represent that, in the sequence of action commands \(\text{actions}(\text{run},n,t2)\), by the no action \text{skip} command:

\[
\text{value}
\]

\[
\text{actions}: \text{RUN} \times N \times T \sim AL
\]

\[
\text{actions}(\text{run},n,t2) \equiv
\]

\[
\text{let } t1:T \cdot \exists (n',l):(N \times L) \cdot (n,l) \in \text{lics}(\text{run})(t1) \text{ in}
\]

\[
\langle (n,a) | a',A,ti:T \cdot ti \leq t1 \land (n,a') \in \text{acts}(\text{run})(t) \Rightarrow a=a' \land (n,a') \notin \text{acts}(\text{run})(t) \Rightarrow a=\text{skip} \rangle
\]

end

\[
\pre: \exists t:T,l:L \cdot t \in \text{dom run} \land t < t2 \land (n,l) \in \text{lics}(\text{run})(t)
\]

Annotations

• Time \(t1\) is the time at which a license named time \(n\) is introduced (into the run \(\text{run}\)).
• Time ti ranges from and including t1 to and including t1.
• For every such time there is a contribution (n,a) to the action sequence yielded by actions(run,n,t2).
• If there is an n related action a issued at time ti, i.e., if there is a pair (n,a) in acts(run)(t),
  then a is the ith elements of actions(run,n,t2),
• else skip is the ith elements of actions(run,n,t2).
• If there does not exist a time t1 in run run at which a license named n is introduced (with time t1 before time t2) then actions(run,n,t2) is undefined.

Permission Observer on Runs

A run both captures the actions of producers and of consumers. The former are captured by the trace of the first, the licenses component of the time-varying configuration. The latter by the second, the actions component of those configurations.

At any one time the sequence of sets of licenses issued by producers up to that time reflects the actions that are permitted hereafter by consumers.

We can defined a function P which when applied to a run run at time t yields a set of pairs, (n,a), of license names and action commands. The idea is that any action a wrt. some license name n being issued by a consumer at time t is allowed if (n,a) is in P(t)(t).

We first define two auxiliary functions. One, is_active, when applied to a run, run, a license name, n and a time t, yields truth if at that or some earlier time a license of the name n has been issued. The other, get_L, when applied to a run, run, a license name, n and a time t, yields the license, l, that is active.

value
is_active: RUN × N × T → Bool
is_active(run,n,t) ≡ ∃ t':T • t' ≤ t ∧ ∃ l:L • (n,l) ∈ lics(run)(t)

get_L: RUN × N × T → L
get_L(run,n,t) ≡
  let t':T • t' ≤ t ∧ ∃ l:L • (n,l) ∈ lics(run)(t) in
  let (n',l):(N×L) • n'=n ∧ (n,l) ∈ lics(run)(t) in (n,l) end end

Then we define P:

value
P: RUN → T → (N×A)-set
P(run)(t) ≡
  {(n,a)|a:A,n:N •
    case is_active(run,n,t) of
    false→ a=skip,
    true→ let (n,l) = get_L(run,n,t) in viable(actions(run,n,t)˘(a),l) end
  end}
3.2.7 The Proof System

Meaning of Action Expression

This is the A of column 1 of page 4 of the Pucella/Weissman paper.

Proof Rules

This is a mere repetition of the proof rules of columns 1 and 2 of page 4 of the Pucella/Weissman paper — albeit augmented with a narrative English “reading” of each of these rules.

Properties of the Proof System

This is to be a summary of the rest of the Pucella/Weissman paper as from column 2 of page 4 onwards.

3.2.8 Discussion

3.3 A CafeOBJ Specification

Authors

This very preliminary and incomplete CafeOBJ model was provided by the fourth author, and it is being worked on by the first and the last two authors (as of Apr 27, 2006).

Notice

Since temporal logic has been introduced into Pucella’s model, which currently is not supported by CafeOBJ, the authors will take different strategy to finish the CafeOBJ model compared with last one, i.e., Gunter’s model. First, the authors will finish the specification mainly based on the RAISE specification (Section 2, which is kindly provided by the second author) at the syntactic level, i.e., focusing on the signature part of the specification and putting the axioms part (equations to implement the semantics) afterwards. After that, the authors will try to develop some modules to implement the temporal semantics. Though in Pucella’s model, only a part of temporal operators (ALWAYS and NEXT) and semantics are introduced, it could be a heavy load to implement it correctly and completely in CafeOBJ in a hurry. A possible aid and useful reference is LTL model check module of Maude, a sibling language of CafeOBJ (as of Apr 27, 2006).

```mod* WORK {
  [ Work ]
  op _=_ : Work Work -> Bool```
var \( W : \text{Work} \)
\[ \text{eq } (W = W) = \text{true} . \]

-- -------------------------------------

mod* DEVICE {
[ Device ]
op \_=\_ : Device Device \to \text{Bool}

var D : Device
\[ \text{eq } (D = D) = \text{true} . \]

-- -------------------------------------

mod* ACTION {
pr (\text{WORK} + \text{DEVICE} + \text{NAT}) -- the amount of money is represented by Nat

[ Action ]

-- constructors of Action
op pay : Nat \to \text{Action} \{ \text{constr} \}
op render : Work Device \to \text{Action} \{ \text{constr} \}
op skip : \to \text{Action} \{ \text{constr} \}
-- It should be noticed that the current CafeOBJ system, however,
-- does not take any advantage of the constructor concept in any
-- way. To the system, \text{constr} is a comment.
}

-- -------------------------------------

mod* NAME { -- license name
[ Name ]

-- An event is the occurrence of a license named action
mod* EVENT {
pr (2TUPLE ( C1 \leq \text{view to NAME} \{ \text{sort Elt} \to \text{Name} \},
C2 \leq \text{view to ACTION} \{ \text{sort Elt} \to \text{Action} \} )
\star \{ \text{sort 2Tuple} \to \text{Event},
op 1* \_ \to n \_ ,
op 2* \_ \to a \_ ,
op \langle \_ ; \_ \rangle \to \_ \& \_ \} )
-- 2TUPLE is built-in module of CafeOBJ
-- 1* \_ : get the first element of 2Tuple
-- 2* \_ : get the second element of 2Tuple

op ce : Event \to Event -- event complement

vars A1 A2 : Action
var n : Name
ceq ce(n & A1) = (n & A2) if not A1 == A2 .
}

-- -------------------------------------
mod* LICENSE {
    pr ( ACTION )

    [ License ]

    -- constructors
    op mkAL : Action -> License { constr }
    op mkCL : License License -> License { constr } -- concatenation
    op mkLC : License -> License { constr } -- closure, *
    op mkUL : License License -> License { constr } -- union
}

-- Module of the modal language of propositions
mod* MPROP {
    pr ( LICENSE + NAME + EVENT )

    [ MProp ]

    -- constructors
    op mkNL : Name License -> MProp { constr }
    op mkNA : Event -> MProp { constr }
    op mkPA : Action -> MProp { constr } -- action is permitted
    op mkOA : Action -> MProp { constr } -- action is obligated
    op _/\_ : MProp MProp -> MProp { constr } -- logic operator And
    op _\/\_ : MProp MProp -> MProp { constr } -- Or
    op !_ : MProp -> MProp { constr } -- Negation
    op Nxt_ : MProp -> MProp { constr } -- temporal logic operator Next
    op Alw_ : MProp -> MProp { constr } -- Always
    op Uti_ : MProp -> MProp { constr } -- Until
}

-- INCOMPLETE, keep revising and updating!

3.4 Comparisons
Chapter 4

The ODRL Paper

4.1 A Transcript of the ODRL Paper

This transcript was kindly provided by the fourth author. More editing work will be done.

4.1.1 Motivation

Remove Ambiguity of ODRL

Problem: ODRL does not have formal semantics, and thus agreements written in ODRL are open to interpretation.

- Example 1:
  Statement: “the group comprised of Alice and Bob is permitted to withdraw money from the bank account BA”

  1. Either Alice or Bob may access the account?
  2. Alice and Bob together may withdraw the money, although neither has permission to do so alone?
  3. The group refers to some third individual, perhaps someone who Alice and Bob both trust?

Problems

- Is it a good example and fair argument?
  * Is this a problem of ODRL or the statement per se (ambiguity)?
  * XML Schema (DTD): Type System
    XML Query and XPath: Formal Semantics
Formal Reasoning of Licenses

Whether a permission is implied by a set of ODRL statements

• Queries: May subject $s$ do action $act$ to asset $a$?
  * Query inconsistent: $\text{Permitted}(s, act, a)$ and $\neg \text{Permitted}(s, act, a)$ both hold
  * Permission granted: only $\text{Permitted}(s, act, a)$ hold
  * Permission denied: only $\neg \text{Permitted}(s, act, a)$ hold
  * Permission unregulated: Neither holds

Problems

• Is it possible to observe license evolution?

4.1.2 The ODRL Language

Example of ODRL Agreement

• ODRL agreement example

```xml
<agreement>
  <asset> <uid> Theasure Island </uid> </asset>
  <permission>
    ...<display>
      <constraint>
        <cpu> <uid> Mary’s computer </uid> </cpu>
      </constraint>
    </display>
    <print>
      <constraint> <count> 2 </count> </constraint>
    </print>
    <requirement>
      <prepay> ... </prepay>
    </requirement>
  </permission>
  <party> <name> Mary Smith </name> </party>
</agreement>
```

• Abstract Syntax of ODRL

```
agreement
  for Mary Smith
  about Treasure Island
  with prePay[5.00] → and [cpu[Mary’s Computer]] → display,
```
Abstract syntax for ODRL

- Abstract Syntax for ODRL (agreements)

\[ agr ::= \text{agreement} \]
\[ \text{agreement} \]
\[ \text{for prin}_u \]
\[ \text{about a} \]
\[ \text{with} ps \]
\[ ps ::= \text{policy set} \]
\[ prq \rightarrow p \] \text{primitive policy set}
\[ prq \rightarrow p \] \text{primitive exclusive policy set}
\[ \text{and}[ps_1,\ldots,ps_m] \] \text{conjunction} \( m \geq 1 \)
\[ p ::= \text{policy} \]
\[ prq \Rightarrow id \ act \] \text{primitive policy}
\[ \text{and}[p_1,\ldots,p_m] \] \text{conjunction} \( m \geq 1 \)
\[ id \in PolIds \] \text{policy identifier}

- Abstract Syntax for ODRL (prerequisites)

\[ prq ::= \text{prerequisite} \]
\[ true \] \text{constraint}
\[ req \] \text{requirement}
\[ cond \] \text{condition}
\[ \text{and}[prq_1,\ldots,prq_m] \] \text{conjunction} \( m \geq 1 \)
\[ \text{or}[prq_1,\ldots,prq_m] \] \text{disjunction} \( m \geq 1 \)
\[ \text{xor}[prq_1,\ldots,prq_m] \] \text{exclusive disjunction} \( m \geq 1 \)

Problems

- What’s the difference between \( \rightarrow \) and \( \Rightarrow id \)?

- Example 1
  \[ \text{agreement for} \{ Alice, Bob \} \text{ about The Report with} [p_1,P_2] \]
  where \( p_1 ::= \text{count}[5] \Rightarrow id_1 \) print and \( p_2 ::= \text{and}[Alice,\text{count}[2]] \Rightarrow id_2 \) print

- Example 2
  \[ \text{agreement for} \{ Alice, Bob \} \text{ about The Report with} \text{count}[5] \rightarrow [p_1,P_2] \]
  where \( p_1 ::= \text{print}, p_2 ::= \text{display} \)

4.1.3 A Semantics in First-Order Logic

Vocabulary

- Permitted: \( Subjects \times Actions \times Assets \rightarrow \text{Bool} \)
- Paid: \( Reals \times SetPolIds \times Times \rightarrow \text{Bool} \)
• **Attributed**: $Subjects \times Times \rightarrow \text{Bool}$

• play display print : constants of sort $Actions$

• $count : Subjects \times PolIds \rightarrow \text{Reals}$

**Variables**

$s : Subjects \quad act : Actions \quad a : Assets$

$r : \text{Reals} \quad I : \text{SetpolIds} \quad t : \text{Time} \ (t < \infty)$

**Translation Models**

• Goal

An agreement of ODRL is translated into a conjunction of formulas:

$$\forall x (\text{prerequisites}(x) \Rightarrow P(x))$$

$$P(x) \triangleq \text{prerequisites}(x) \Rightarrow (\neg \text{Permitted}(x, act, a))$$

• Translation of ODRL agreement

$$[[\text{agreement for prin_a about a with ps}]] \triangleq [[ps]]_{prin_a, a}$$

$$[[prq \rightarrow p]]_{prin_a, a} \triangleq \forall x ([[prin_a]]_x \land [prq]_{ids(p), prin_a, a}^{x}) \Rightarrow [p]_{x, prin_a, a}^{x}$$

$$[[prq \rightarrow p]]_{prin_a, a} \triangleq \forall x ([[prin_a]]_x \land [prq]_{ids(p), prin_a, a}^{x}) \Rightarrow [p]_{x, prin_a, a}^{x} \land \forall x (\neg [[prin_a]]_x \Rightarrow [p]_{x, a}^{x})$$

$$[[prq \rightarrow i\ dact]]_{prin_a, a}^{x} \triangleq ([prq]_{x}^{(id), prin_a, a}) \Rightarrow \text{Permitted}(x, [act], a)$$

• Translation of ODRL prerequisites

$$[[\text{true}]]_{x, prin_a, a} \triangleq \text{true}$$

$$[[\text{prin}]]_{x, prin_a, a} \triangleq [[\text{prin}]]_x$$

$$[[\text{forEachMember}[\text{prin}; cons_1, \ldots, cons_m]]_{x, prin_a, a} \triangleq \land_{(prin', i) \in P_m} [\text{cons_1}^{x, prin_a, a}, \ldots, \text{cons_m}^{x, prin_a, a}], \text{where}$$

$$P_m = \text{principals}(\text{prin}) \times \{1, \ldots, m\}$$
\[
\exists l_{i+1,prin,a} \triangleq (\Sigma_{(id,s) \in I \times (subjects(prin))} count(s, id)) < n
\]
\[
\exists l_{i+1,prin,a} \triangleq (\Sigma_{(id,s) \in I \times (subjects(prin))} count(s, id)) < n
\]

- Example of Translation

* Example of ODRL Agreement

\[
\text{agreement}
\]
\[
\text{for } \{\text{Alice, Bob}\}
\]
\[
\text{about } \text{ebook}
\]
\[
\text{with } \text{count}[10] \rightarrow
\]
\[
\text{and}[\text{forEachMember}[\{\text{Alice, Bob}\}; \text{count}[5]] \Rightarrow \text{id1 display}
\]
\[
\text{forEachMember}[\{\text{Alice, Bob}\}; \text{count}[1]] \Rightarrow \text{id2 print}
\]

* Translation of First-Order Logic Formula

\[
\forall x((x = \text{Alice} \lor x = \text{Bob}) \Rightarrow
\]
\[
\text{count}(\text{Alice, id1}) + \text{count}(\text{Alice, id2}) +
\]
\[
\text{count}(\text{Bob, id2}) + \text{count}(\text{Bob, id2}) \leq 10 \Rightarrow
\]
\[
((\text{count}(\text{Alice, id1}) < 5 \land \text{count}(\text{Bob, id2}) < 5) \Rightarrow
\]
\[
\text{Permitted}(x, \text{display, ebook})) \land
\]
\[
((\text{count}(\text{Alice, id2}) < 1 \land \text{count}(\text{Bob, id2}) < 1) \Rightarrow
\]
\[
\text{Permitted}(x, \text{print, ebook}))
\]

4.1.4 Queries

- Environment

An Environment is a conjunction of positive ground literals, each of the form \text{Attributed}(s, t)

or \text{Paid}(s, I, t), and equalities of the form \text{count}(s, id) = n.

- Goal

Reasoning whether a set \( A \) of agreements imply that a subject \( s \) may do action \( act \) to asset \( a \) in environment \( E \).

```
- Does the word \text{Permitted}-free ground literal refer to the literal which is not \text{Permitted}, such as \text{Attributed}, \text{Paid}, and \text{count}?
```

- E-validity of Queries

Suppose a query \( q = (A, s, act, a, E) \)

\[
f^+_q \triangleq \bigwedge_{agr \in A} [agr] \Rightarrow \text{Permitted}(s, act, a)
\]
\[ f_q^- \triangleq \bigwedge_{agr \in A} [agr] \Rightarrow \neg \text{Permitted}(s, act, a) \]

* Query Inconsistent: Both \( f_q^+ \) and \( f_q^- \) are E-valid.
* Permission Granted: Only \( f_q^+ \) is E-valid.
* Permission Denied: Only \( f_q^- \) is E-valid.
* Permission Unregulated: Neither \( f_q^+ \) nor \( f_q^- \) is E-valid.

- **Example**
  Suppose \( A = agr, agr' \), where \( agr \) is

  **agreement for Alice about file with print**

  and \( agr' \) is

  **agreement for Bob about file with true \( \rightarrow \) print**

- **Problems**
  - The result of the last query above should be Permission Denied rather than Permission Unregulated? A typo or misunderstanding?

### 4.1.5 Summary

- **Formal Reasoning of agreements of ODRL.**
- Do not describe how to update environments, but to compute which actions are permitted in any given environment.
  - Holzer et al. (2004): Towards a formal semantics for ODRL.
  - OTS/CafeOBJ may be used to describe both license evolution and verification.
- **Study formal RELs with practical RELs such as ODRL and XrML**
- Related reference: Pucella (2004): Using first-order logic to reason about policies
4.2 A RAISE/RSL Specification

4.3 A CafeOBJ Specification

This very preliminary and incomplete CafeOBJ model was provided by the fourth author, and it is being worked on by the first and the last two authors (as of Jun29, 2006).

```
mod* SET (X :: TRIV) {  
  pr (EQL)  
  [ Set ]  
  op empty : -> Set  
  op add : Elt Set -> Set  
  op _in_ : Elt Set -> Bool  
  vars E E' : Elt  
  var S : Set  
  eq E in add(E', S) = (E = E') or (E in S) .  
  eq E in empty = false .  
  eq (empty = add(E,S)) = false .  
}  

mod* SET+ {  
  pr (SET)  
  op _+_ : Set Set -> Set { assoc comm prec: 37 }  
  vars E E' : Elt.X.SET  
  vars S S' : Set  
  eq empty + empty = empty .  
  eq empty + add(E,S) = add(E,S) .  
  eq add(E,S) + add(E',S')  
  = (if E in add(E',S')  
     then S + add(E',S')  
     else add(E, (S + add(E',S'))) fi) .
}  

mod* 2SET (S :: SET, S' :: SET ) {  
  pr (SET* *( sort Elt -> 2Elt,  
           sort Set -> 2Set,  
           op empty -> 2empty))  
  op _*_ : Set.S Set.S' -> 2Set { prec: 35 }  
  op <<;,:>> : Elt.S Elt.S' -> 2Elt
```
vars E1 E2 : Elt.S
vars E1' E2' : Elt.S'
var S : Set.S
var S' : Set.S'
var 2S : 2Set
var 2E : 2Elt

eq empty.S * empty.S' = 2empty .
eq empty.S * add(E1',S') = 2empty .
eq add(E1,S) * empty.S' = 2empty .
eq add(E1,S) * add(E1',S')
  = add(<< E1 ; E1' >> , add(E1,S) * S') + (S * add(E1',S')) .
eq (<< E1 ; E1' >> = << E2 ; E2' >>)
  = (E1 = E2) and (E1' = E2') .

-- eq (2empty = add(2E,2S)) = false .
}

-- -- test
-- open 2SET
--
-- ops e1 e2 : -> Elt.S .
-- ops e1' e2' : -> Elt.S' .
-- op s : -> Set.S .
-- op s' : -> Set.S' .
--
-- eq (e1 = e2) = false .
-- eq (e1' = e2') = false .
--
-- red add(e1, add(e2, empty.S)) * add(e1', add(e2', empty.S')) .
-- close
-- eof
-- --

mod* ASSET {
  pr (EQL)

  [ Asset ]
}

mod* SUBJECT {
  pr (EQL)

  [ Subject ]
}

mod* SUBSET {
  pr (SET+(SUBJECT { sort Elt -> Subject })
    *( sort Set -> SubSet ,
      op empty -> em-subset )} }

mod* PRINCIPAL {
  -- pr (SUBJECT)
  -- pr (SUBSET)

-- Here we define a principal is a set of subject-sets
pr (SET+(SUBSET { sort Elt -> SubSet }))
  *( sort Set -> Prin,
    op empty -> em-prin ))
pr (SUBJECT)

op subjects : Prin -> SubSet

vars S S' : Subject
vars X X' : SubSet
vars P P' : Prin

eq subjects(em-prin) = em-subset .
eq subjects(add(add(S,X),P))
  = (if (X = em-subset)
    then add(S,subjects(P))
    else add(S,subjects(add(X,P))) fi) .

mod* ACTION {
  pr (EQL)

    [ Action ]

    ops play print display : -> Action

    eq (play = print) = false .
eq (play = display) = false .
eq (print = display) = false .
}

mod* POLID {
  pr (EQL)

    [ PolId ]
}

mod* CONSTRAINT {
  pr (EQL)

    [ Cons ]
}

mod* CONSTRAINTSET {
  pr (SET(CONSTRAINT{ sort Elt -> Cons }))
    *( sort Set -> ConsSet ,
      op empty -> em-consset ))
}

mod* CONS {
  pr (CONSTRAINTSET)
  pr (PRINCIPAL + NAT)

  -- user constraint
  op user : Prin -> Cons
  op forEachMember[_;_] : Prin ConsSet -> Cons

  -- count constraint
\begin{verbatim}
  op count[_.]  : Nat -> Cons
  op _<count[_.]> : Prin Nat -> Cons

  mod* REQUIREMENT {
    pr (EQL)
    [ Req ]
  }

  mod* REQUIREMENTSET {
    pr (SET(REQUIREMENT)
      *( sort Set -> ReqSet ,
        op empty -> em-regset ))
  }

  mod* REQ {
    pr (REQUIREMENTSET)
    pr (SUBJECT + RAT)
    op prePay[_.]  : Rat -> Req
    op attribution[_.] : Subject -> Req
    op inSeq[_.] : ReqSet -> Req
    op anySeq[_.] : ReqSet -> Req
  }

  -- sort 'PolicySet' must be defined before sort 'Cond', since the
  -- 'not' constructor of 'Cond' requires a parameter of 'Policyset'.
  mod* POLICYSET {
    pr (EQL)
    [ PolicySet ]
  }

  mod* COND {
    pr (POLICYSET + CDNS)
    [ Cond ]
    op not[_.] : PolicySet -> Cond
    op not[_.] : Cons -> Cond
  }

  mod* PRQ {
    pr (CDNS + REQ + COND)
    [ Cons Req Cond < Prq ]
    op True : -> Prq
    op _and_ : Prq Prq -> Prq
    op _or_ : Prq Prq -> Prq
    op _xor_ : Prq Prq -> Prq
  }

  mod* POLICY {
    pr (EQL)
  }
\end{verbatim}
pr (PRQ + POLID + ACTION)

[ Policy ]
op _==>_ : Prq PolId Action -> Policy
op _and_ : Policy Policy -> Policy
}

-- Defining constructors of 'PolicySet' though the sort has been
defined beforehand.
mod* PS {
pr (POLICYSET + POLICY)

op _~>_ : Prq Policy -> PolicySet
op _|~>_ : Prq Policy -> PolicySet
op _and_ : PolicySet PolicySet -> PolicySet
}

mod* AGREEMENT {
pr (PRINCIPAL + ASSET + PS)

[ Agr ]
op agreement-for_about_with_ : Prin Asset PolicySet -> Agr
}

-- ---------------------------------------------------------------
-- ---------------------------------------------------------------
-- Translation of ODRL agreements, i.e., semantics (axioms) of the
-- agreements.
-- ---------------------------------------------------------------

mod* SETPOLID {
pr (SET+(POLID)
    *( sort Set -> SetPolId ,
      op empty -> em-pids ))
    -- pr (AGREEMENT)
--
-- op ids : Policy -> SetPolId
--    -- op ids : PolicySet -> SetPolId
--    -- op pids : Policy -> SetPolId
--
--    -- vars ID ID1 ID2 : PolId
--    -- vars P P1 P2 : Policy
--    -- vars PS PS1 PS2 : PolicySet
--    -- var PRQ : Prq
--    -- var ACT : Action
--
--    -- eq ids(PRQ ~> P) = pids(P) .
--    -- eq ids(PRQ |~> P) = pids(P) .
--    -- eq ids(PS1 and PS2) = ids(PS1) + ids(PS2) .
--
--    -- eq pids(PRQ ==> ID ACT) = add(ID, em-pids) .
--    -- eq pids(P1 and P2) = pids(P1) + pids(P2) .
--    -- eq ids(PRQ ==> ID ACT) = add(ID, em-pids) .
--    -- eq ids(P1 and P2) = ids(P1) + ids(P2) .
June 29, 2006 — Y.Arimoto, D.Bjørner, X.Chen, J.Xiang

---

-- -- test
-- open SETPOLID
-- ops id1 id2 id3 id4 : -> PolId.
-- ops prq1 prq2 prq3 prq4 prq : -> Prq.
-- ops act1 act2 act3 act4 : -> Action.
-- ops ps1 ps2 : -> PolicySet.
-- ops p1 p2 p3 p4 : -> Policy.
--
-- eq (id1 = id2) = false.
-- eq (id1 = id3) = false.
-- eq (id1 = id4) = false.
-- eq (id2 = id3) = false.
-- eq (id2 = id4) = false.
-- eq (id3 = id4) = false.
--
-- eq ps1 = prq1 ~> p1.
-- eq ps2 = prq2 |~> p2.
-- eq p1 = p3 and p4.
-- eq p2 = prq2 ==> id2 act2.
-- eq p3 = prq3 ==> id3 act3.
-- eq p4 = prq4 ==> id4 act4.
--
-- red ids(ps1 and ps2).
-- close
-- ecf
-- -- -----------------------------------------------

mod* TIME { 
pr (RAT)

[ Rat < Time ]

op oo : -> Time -- infinity
}

-- Agreement Formulas
mod* AGR-F {
pr (AGREEMENT + SETPOLID + RAT + TIME)
pr (2SET(S <= SETPOLID { sort Elt -> PolId, 
sort Set -> SetPolId, 
op empty -> em-pids, 
op add -> add, 
op _in_ -> _in_ }),
S' <= SUBSET { sort Elt -> Subject, 
sort Set -> SubSet, 
op empty -> em-subset, 
op add -> add, 
op _in_ -> _in_ })
*( sort 2Set -> IdSub, 
sort 2Elt -> Id-Sub 
op 2empty -> 2em-is })
pr (2SET(S <= CONSTRAINTSET { sort Elt -> Cons, 
sort Set -> ConsSet, 
op empty -> em-consset,
op add -> add,  
op _in_ -> _in_,  
S' <= PRINCIPAL { sort Elt -> SubSet,  
sort Set -> Prin,  
op empty -> em-prin,  
op add -> add,  
op _in_ -> _in_ }  
*( sort 2Set -> Cons*Prin,  
sort 2Elt -> Cons-Prin  
op 2empty -> 2em-cp })

-- predicates  
op permitted : Subject Action Asset -> Bool  
op paid : Rat SetPolId Time -> Bool  
op attributed : Subject Time -> Bool

-- functions  
op count : Subject PolId -> Rat  
op sum-count : Id*Sub -> Rat  
op and-cons : Cons*Prin SetPolId Asset Subject -> Bool  
op ids : Policy -> SetPolId

op F : Agr Subject -> Bool  
op F : PolicySet Prin Asset Subject -> Bool  
op F : Policy Prin Asset Subject -> Bool  
op F : Policy Asset Subject -> Bool  
op F : Policy Prin SetPolId Asset Subject -> Bool  
op F : Prin Subject -> Bool  
op F : Prq SetPolId Prin Asset Subject -> Bool  
op F : Req SetPolId Time Time -> Bool

-- variables  
var A : Asset  
var ACT : Action  
vars PRIN PRIN' : Prin  
vars S X : Subject  
vars PS PS1 PS2 : PolicySet  
vars P P1 P2 : Policy  
vars PRQ PRQ1 PRQ2 : Prq  
var ID : PolId  
var IDS : SetPolId  
var C : Cons  
var CS : ConsSet  
var N : Nat  
vars T T' T'' : Rat -- time, here use Rat for convenience  
var REQ : Req  
var R : Rat  
var I*S : Id*Sub  
var C*P : Cons*Prin  
var SS : SubSet

eq ids(PRQ == ID ACT) = add(ID, em-pids) .  
eq ids(P1 and P2) = ids(P1) + ids(P2) .  

eq sum-count(add(< ID ; S >>, I*S ))  
  = (if I*S = 2em-is then count(S,ID)
else (count(S,ID) + sum-count(I*S)) fi).

eq \text{and-cons}(\text{add}(\text{<< C ; SS >>}, C*P),IDS,A,X)
= (\text{if } C*P = 2\text{em-cp}
\text{then } F(C,IDS,\text{add}(SS,em-prin),A,X)
\text{else } F(C,IDS,\text{add}(SS,em-prin),A,X)
\text{and and-cons}(C*P,IDS,A,X)
\text{fi}).

-- equations for translation of ODRL agreement, corresponding to
-- Fig. 3 of the paper

eq F(\text{agreement-for (PRIN) about (A) with (PS),X}) = F(PS,PRIN,A,X) .

eq F(PRQ \Rightarrow P,PRIN,A,X)
= (F(PRIN,X) \text{ and } F(PRQ,ids(P),PRIN,A,X)) \Rightarrow F(P,PRIN,A,X) .
eq F(PRQ \Rightarrow P,PRIN,A,X)
= (F(PRIN,X) \text{ and } F(PRQ,ids(P),PRIN,A,X)) \Rightarrow F(P,PRIN,A,X)
\text{and (not } F(PRIN,X) \Rightarrow \text{not } F(P,A,X)) .
eq F(PS1 \text{ and } PS2,PRIN,A,X) = F(PS1,PRIN,A,X) \text{ and } F(PS2,PRIN,A,X) .

eq F(PRIN,X) = X \text{ in subjects(PRIN)} .
-- eq F(S,X) = (X = S) .

eq F(PRQ \Rightarrow ID ACT,PRIN,A,X)
= F(PRQ,add(ID,em-pids),PRIN,A,X) \Rightarrow \text{permitted}(X,ACT,A) .
eq F(PRQ \Rightarrow ID ACT,PRIN,A,X)
= F(PRQ,add(ID,em-pids),PRIN,A,X) \Rightarrow \text{permitted}(X,ACT,A)
\text{and (not } F(PRIN,X) \Rightarrow \text{not } F(P,A,X)) .

-- here we omit F(ACT) = ACT as listed in the paper, since we
-- directly wrote \text{permitted}(X,ACT,A) instead of
-- \text{permitted}(X,F(ACT),A) in the above equations.

-- equations for translation of ODRL prerequisites, Fig. 4

eq F(True,IDS,PRIN,A,X) = true .
eq F(user(PRIN'),IDS,PRIN,A,X) = F(PRIN',X) .

eq F(forEachMember[ PRIN' ; CS ],IDS,PRIN,A,X)
= \text{and-cons}(CS * PRIN',IDS,A,X) .

eq F(count[ N ],IDS,PRIN,A,X)
= \text{sum-count}(IDS * subjects(PRIN)) \text{ < } N .
eq F(\text{prePay[ R ]},IDS,T,T')
= \text{paid}(R,IDS,T'') \text{ and } (T < T'') \text{ and } (T'' < T') .
-- here omit the attribution, inSeq, and anySeq for laziness

-- eq F(not[ PS ],IDS,PRIN,A,X) = not F(PS,PRIN,A) .
-- eq F(not[ PS ],IDS,PRIN,A,X) = not F(PS,PRIN,A,X) .
-- eq F(not[ C ],IDS,PRIN,A,X) = not F(C,IDS,PRIN,A,X) .

-- the following three eq. have parameter A which does not appear in
-- the paper, a typ or my misunderstanding?
eq F(PRQ1 \text{ and } PRQ2,IDS,PRIN,A,X)
\[ F(PRQ1, IDS, PRIN, A, X) \text{ and } F(PRQ2, IDS, PRIN, A, X) \]
\[ = F(PRQ_1 \text{ or } PRQ_2, IDS, PRIN, A, X) \]
\[ = F(PRQ_1, IDS, PRIN, A, X) \text{ or } F(PRQ_2, IDS, PRIN, A, X) \]
\[ = F(PRQ_1 \text{ xor } PRQ_2, IDS, PRIN, A, X) \]

-- test
open AGR-F

ops Alice Bob : \rightarrow Subject.
op ebook : \rightarrow Asset.
ops id1 id2 : \rightarrow PolId.
ops p p1 p2 : \rightarrow Prin.
ops a1 a2 : \rightarrow Agr.
ops cs1 cs2 : \rightarrow ConsSet.

var X : Subject

eq (Alice = Bob) = false.
eq (id1 = id2) = false.
eq p = add(add(Bob,em-subset),add(add(Alice,em-subset),em-prin)).
eq p1 = add(add(Alice,em-subset),em-prin).
eq p2 = add(add(Bob,em-subset),em-prin).
eq cs1 = add(count[5],em-consset).
eq cs2 = add(count[1],em-consset).
eq a1 = agreement-for (p)
\hspace{1em} about (ebook)
\hspace{2em} with (count[10] \rightarrow (p1 <count[5]> \Rightarrow id1 display) and
\hspace{3em} (p2 <count[1]> \Rightarrow id2 print)).
eq a2 = agreement-for (p)
\hspace{1em} about (ebook)
\hspace{2em} with (count[10] \rightarrow
\hspace{3em} (forEachMember[p; cs1] \Rightarrow id1 display) and
\hspace{4em} (forEachMember[p; cs2] \Rightarrow id2 print)).

-- Notice the result looks different with the examples in the paper,
-- because the logic operator "implies" in the BOOL module is
-- translated into "xor".

-- red F(a1,X).
-- red F(a2,X).

-- Environment and Query

-- setting initial values to the Environment

-- An environment of agreements is a conjunction of positive ground
-- literals, each of the form Attributed(s,t) or Paid(s,I,t), and
-- equalities of the form count(s,id) = n.

-- Here we model Environment as a set of equations setting initial
-- values for the predicates Attributed, Paid, and count as follows:

eq \text{count}(\text{Alice}, \text{id1}) = 3 .
eq \text{count}(\text{Bob}, \text{id1}) = 3 .
eq \text{count}(\text{Alice}, \text{id2}) = 1 .
eq \text{count}(\text{Bob}, \text{id2}) = 0 .

-- Environment setting finished

-- Query check

\text{red } F(\text{a2}, X) \text{ and } (X = \text{Alice}) \text{ implies permitted}(X, \text{display}, \text{ebook}) . \text{ -- return true since } 3 < 5 \text{ and } 3 + 3 + 1 + 0 < 10

\text{red } F(\text{a2}, X) \text{ and } (X = \text{Alice}) \text{ implies permitted}(X, \text{print}, \text{ebook}) . \text{ -- should return false since } ((\text{count}(\text{Alice}, \text{id2}) = 1) < 1) = \text{false}

-- however, the system does not return false as we wanted!!!

-- The problem is that if we understood the result of $F(\text{a2}, X)$ in
-- a form of "A -> C", in which A denotes the term consisting of
-- non-permitted predicates while C denotes the term consisting
-- of a conjunction of permitted predicates, then the above
-- reasoning can be treated as "(A -> C) -> Q?" where Q is the
-- query, i.e., permitted(X,print,ebook) in the above command.

-- Here, since A is wrong (false), so the result is Q, a Boolean
-- term instead of false as we wanted!

-- A bug of the paper?

-- Possible Solution: Separating $F(\text{a2}, X)$ into two terms, i.e.,
-- A and C, and then reducing in a form of:

-- "red (A and (C implies Q))."

-- Return: false, then Q does not hold (Permission denied);
-- Return: true, then Q holds (Permission granted);
-- Return: a Boolean term, then Q is unknown (Permission unregulated);
-- As for "Query inconsistent", we need only check whether C is false,
-- i.e., one agreement grant the permission, while the other use
-- exclusive policy denied the permission with respect to a specific
-- principal. C then contains "P and not P".

\text{red } F(\text{a2}, \text{Bob}) \text{ and } (X = \text{Bob}) \text{ implies permitted}(X, \text{display}, \text{ebook}) . \text{ -- return true}

\text{red } F(\text{a2}, X) \text{ and } (X = \text{Bob}) \text{ implies permitted}(X, \text{print}, \text{ebook}) . \text{ -- should return false since \text{count}(\text{Alice}, \text{id2}) = 1, though}

-- count(\text{Bob}, \text{id2}) = 0 < 1. Recalling the def. of forEachMember
-- the same problem as the second red.

close
eof
4.4 Comparisons
Chapter 5

The XrML Paper
Chapter 6

The DRM Reference Model Paper

<table>
<thead>
<tr>
<th>Intentions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB intends to model the “Long March” paper on a “ReferenceModel of Digital Rights Management” Issues to be modelled: Subsystems, their interfaces, and hence the issues of interoperability and trusted subsystems.</td>
</tr>
</tbody>
</table>
Chapter 7

Digital Rights and Game Theory

7.1 Preventing Secrets Sharing in Online Subscription Services

A possible application of GT for discussion

The problem originates from password (or secrets) sharing and misuse in online subscription services, which we have proposed a protocol called IBPS (Incentives Based Secrets Protection System) to solve this problem (see http://www.ldl.jaist.ac.jp/drcp/doc/IBPS.pdf). The paper (which has been accepted by WEBIST’06, http://www.webist.org/) is a preliminary work on the topic of incentives-based method to prevent password sharing, and it has only briefly discussed the strategies of producers and consumers based on simple mathematical algorithms and analysis. Some future work and more comprehensive analysis especially combined with RGT (rewriting game theory) MAY further improve it. Perhaps it is just a stupid idea or nonsense, just for discussing the possibility which may cost a little bit time.

7.1.1 Problem Domain

Problem

- Sharing secret among friends becomes a common practice in DRM
  - * Password sharing in digital content subscription services
    - ◦ Napster users sharing passwords to save cash [MusicAlly05]
  - * Domain certificate sharing in Authorized Domain (AD)
  - * Customary personal uses of copyrighted works?
    - ◦ Physical purchased CDs and books Club membership
• Terms of Services in DRM: Each subscriber agrees that not to allow others to use her/his member name, password and/or account

• Password sharing defeats a combination of technical and contractual access control measures [Mulligan03]

Technical Solutions

• Binding account to specific computers ⇒ compromise users portability

• Preventing concurrent access to an account

• Sharing cannot be eradicated even in an extreme case
  * One computer ⇒ sharing the computer and usage time

• Fundamental trade-off between control and customer value [Varian99]

Legal Measures

• Traditional customary personal uses? legally Questionable behaviors [Mulligan03]

• Difficulties to make a plausible claim against password misuse
  * Detect the difference in the physical identity of the user family or friends?
  * Meet respectively high damage threshold

  Computer Fraud and Abuse Act (CFAA): a loss of 5000$ or more in a one-year period ⇒ plaintiffs pursuing a civil claim

• In practice: providers usually prefer contractually reserved self-help measures to laws ⇒ Revoke suspicious accounts

User’s Concerns

• Customary personal uses of copyrighted works?

• Benefits of sharing
  * Earn friendship
  * Save money even a little (youngsters)
  * Lose nothing even concurrent access is prohibited
    ◦ Nobody would like surf 24 hours
    ◦ Spare utility of the password is a good gift for friends

• Harms of sharing
  * Potential legal troubles: an arguable issue and seldom care

• It seems that no good reason for keeping password private
7.1.2 Basic Scheme of IBSPS

- Assumptions
  - Provider
    - Streaming content (movie, music, etc.)
    - Period pricing system
  - Simultaneously assessing is technically prohibited
  - No restriction to binding specific computers (users portability)
- Lottery (bonus): providing positive incentives to keep password private
  - Authorized Consumer: incentive to not share password to win bonus
  - Unauthorized Consumer: incentive to participate the lottery
  - Provider: incentive to provide bonus to attract more authorized consumers

7.1.3 Details of IBSPS

Please see the paper for details.

7.1.4 Possible Advantages of IBSPS

- A positive incentive to keep secret private rather than share among others
- Transform some of the burden on secret protecting into the benefits of customers
- Unexpected password stolen? lose some chance to win the bonus (consumer-oriented)
- Social Significance: cultivating a good habit of not sharing secret

7.1.5 Discussion

- May not work in case simultaneously accessing is not technically prohibited
  - ⇒ A user can share it without losing any utility
  - A technical problem rather than a limitation
  - Even so, IBSPS may still work to some extent if the consumer does not want to share her luck with others
- May fail if sharing both the cost and utility of a secret is more cost-effective wrt. some people
  - ⇒ Some users are not satisfied with current pricing systems
  - More flexible pricing and payment methods should be provided
  - A pricing problem rather than a limitation
- Can not be applied to content files directly
  - It is difficult to define the protection period as for purchased content files
Chapter 8

Conclusion
Chapter 9

Bibliographical Notes

Bibliography


Appendix A

RSL: The RAISE Specification Language

A.1 Types

This is a very brief refresher on the RAISE Specification Language RSL. The reader is kindly asked to study first the decomposition of this section into its subparts and sub-subparts.

A.1.1 TypeExpressions

RSL has a number of built-in types. There are the Booleans, integers, natural numbers, reals, characters and texts. From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc. Let A, B and C be any type names or type expressions, then the following (save the [i] line numbers) are generic type expressions:

<table>
<thead>
<tr>
<th>type</th>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Bool</td>
<td></td>
</tr>
<tr>
<td>[2] Int</td>
<td></td>
</tr>
<tr>
<td>[3] Nat</td>
<td></td>
</tr>
<tr>
<td>[4] Real</td>
<td></td>
</tr>
<tr>
<td>[5] Char</td>
<td></td>
</tr>
<tr>
<td>[8] A-infset</td>
<td></td>
</tr>
<tr>
<td>[9] A × B × ... × C</td>
<td></td>
</tr>
<tr>
<td>[10] A*</td>
<td></td>
</tr>
<tr>
<td>[12] A  m B</td>
<td></td>
</tr>
<tr>
<td>[14] A  B</td>
<td></td>
</tr>
<tr>
<td>[15] (A)</td>
<td></td>
</tr>
</tbody>
</table>
Annotations:

1. The Boolean type of truth values `false` and `true`.
2. The integer type on integers ..., −2, −1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values "a", "b", ....\(^1\)
6. The text type of character string values "aa", "aaa", ..., "abc", ....
7. The set type of finite set values, see below.
8. The set type of infinite set values.
9. The Cartesian type of Cartesian values, see below.
10. The list type of finite list values, see below.
11. The list type of infinite list values.
12. The map type of finite map values, see below.
13. The function type of total function values, see below.
14. The function type of partial function values.
15. In \( (A) \) \( A \) is constrained to be:
   - either a Cartesian \( B \times C \times ... \times D \), in which case it is identical to type expression kind 9,
   - or not to be the name of a built-in type (cf. 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., \( (A \begin{array}{c} \rightarrow \end{array} B) \), or \( (A^*) \)-set, or \( (A-set) \)-list, or \( (A\mid B) \) \( \begin{array}{c} \rightarrow \end{array} (C\mid D\mid(E \begin{array}{c} \rightarrow \end{array} F)) \), etc.
16. The (postulated disjoint) union of types \( A, B, \ldots, \) and \( C \).
17. The record type of \( mk\_id \)-named record values \( mk\_id(av, \ldots, bv) \), where \( av, \ldots, \) and \( bv \) are values of respective types. The distinct identifiers \( sel\_a \), etc., designate selector functions.
18. The record type of unnamed record values \( (av, \ldots, bv) \), where \( av, \ldots, \) and \( bv \) are values of respective types. The distinct identifiers \( sel\_a \), etc., designate selector functions.

\(^1\)RSL uses double quotes " on both sides of a character and a character string rather than usual balanced quotes "..."
A.1.2 Type Definitions

Concrete Types:

Types can be concrete, in which case the structure of the type is specified by type expressions:

```
<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>type A = Type_expr</td>
</tr>
</tbody>
</table>
```

Some schematic type definitions are:

```
<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Type_name = Type_expr</td>
</tr>
<tr>
<td>[2] Type_name = Type_expr_1</td>
</tr>
</tbody>
</table>
| [3] Type_name ==
  mk_id_1(s_a1:Type_name_a1,...,s_aim:Type_name_aim) |
  ...
  mk_id_n(s_z1:Type_name_z1,...,s_zkn:Type_name_zkn) |
| [4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z |
| [5] Type_name = {v:Type_name'} • P(v) |
```

where a form of [2–3] is provided by combining the types:

```
<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type_name = A</td>
</tr>
<tr>
<td>A == mk_id_1(s_a1:A_1,...,s_aim:A_aim)</td>
</tr>
<tr>
<td>B == mk_id_2(s_b1:B_1,...,s_bn:B_bn)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Z == mk_id_n(s_z1:Z_1,...,s_zkn:Z_kn)</td>
</tr>
</tbody>
</table>
```

Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate P constitutes the subtype A:

```
<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>type A = {b:B • P(b)}</td>
</tr>
</tbody>
</table>
```
Sorts (Abstract Types)

Types can be sorts (abstract) in which case their structure is not specified:

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
</tr>
<tr>
<td>A, B, ..., C</td>
</tr>
</tbody>
</table>

A.2 The RSL Predicate Calculus

A.2.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values. Then

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>false, true</td>
</tr>
<tr>
<td>a, b, ..., c</td>
</tr>
<tr>
<td>~a, a&amp; b, a\lor b, a\Rightarrow b, a=b, a\neq b</td>
</tr>
</tbody>
</table>

are propositional expressions having Boolean values. ~, \&, \lor, \Rightarrow, and = are Boolean connectives (i.e., operators). They are read: not, and, or, if-then (or implies), equal and not-equal.

A.2.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values, and let i, j, ..., k designate number values, then

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>false, true</td>
</tr>
<tr>
<td>a, b, ..., c</td>
</tr>
<tr>
<td>~a, a&amp; b, a\lor b, a\Rightarrow b, a=b, a\neq b</td>
</tr>
<tr>
<td>x=y, x\neq y, i&lt;j, i\leq j, i\geq j, i&gt;j, ...</td>
</tr>
</tbody>
</table>

are simple predicate expressions.

A.2.3 Quantified Expressions

Let X, Y, ..., C be type names or type expressions, and let P(x), Q(y) and R(z) designate predicate expressions in which x, y and z are free. Then

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall x:X \cdot P(x)</td>
</tr>
</tbody>
</table>
are quantified expressions — also being predicate expressions. They are “read” as: For all \( x \) (values in type \( X \)) the predicate \( P(x) \) holds; there exists (at least) one \( y \) (value in type \( Y \)) such that the predicate \( Q(y) \) holds; and there exists a unique \( z \) (value in type \( Z \)) such that the predicate \( R(z) \) holds.

### A.3 Concrete RSL Types

#### A.3.1 Set Enumerations

Let the below \( a \)s denote values of type \( A \), then the below designate simple set enumerations:

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}, {a}, {a_1, a_2, ..., a_m}, ... } \in A-set</td>
</tr>
<tr>
<td>{}, {a}, {a_1, a_2, ..., a_m}, ..., {a_1, a_2, ..., } \in A-infset</td>
</tr>
</tbody>
</table>

The expression, last line below, to the right of the \( \equiv \), expresses set comprehension. The expression “builds” the set of values satisfying the given predicate. It is highly abstract in the sense that it does not do so by following a concrete algorithm.

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
</tr>
<tr>
<td>( A, B )</td>
</tr>
<tr>
<td>( P = A \rightarrow \text{Bool} )</td>
</tr>
<tr>
<td>( Q = A \leadsto B )</td>
</tr>
<tr>
<td>value</td>
</tr>
<tr>
<td>comprehend: ( A-infset \times P \times Q \rightarrow B-infset )</td>
</tr>
<tr>
<td>comprehend(s,P,Q) ( \equiv { Q(a) \mid a:A \cdot a \in s \land P(a) } )</td>
</tr>
</tbody>
</table>

#### A.3.2 Cartesian Enumerations

Let \( e \) range over values of Cartesian types involving \( A, B, \ldots, C \) (allowing indexing for solving ambiguity), then the below expressions are simple Cartesian enumerations:

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
</tr>
<tr>
<td>( A, B, ..., C )</td>
</tr>
<tr>
<td>( A \times B \times ... \times C )</td>
</tr>
<tr>
<td>value</td>
</tr>
</tbody>
</table>
A.3.3 List Enumerations

Let \( a \) range over values of type \( A \) (allowing indexing for solving ambiguity), then the below expressions are simple list enumerations:

\[
\{ \langle \rangle, \langle a \rangle, ..., \langle a_1,a_2,...,a_m \rangle, ..., \langle a_1,a_2,...,a_m,a_j \rangle \} \in A^* \\
\{ \langle \rangle, \langle a \rangle, ..., \langle a_1,a_2,...,a_m \rangle, ..., \langle a_1,a_2,...,a_m,a_j \rangle \} \in A^\omega \\
\langle e_i .. e_j \rangle
\]

The last line above assumes \( e_i \) and \( e_j \) to be integer-valued expressions. It then expresses the set of integers from the value of \( e_i \) to and including the value of \( e_j \). If the latter is smaller than the former then the list is empty.

The last line below expresses list comprehension.

\[
\text{type} \\
A, B, P = A \rightarrow \text{Bool}, Q = A \rightsquigarrow B \\
\text{value} \\
\text{comprehend}: A^\omega \times P \times Q \rightsquigarrow B^\omega \\
\text{comprehend}(\text{lst},P,Q) \equiv \\
\{ Q(\text{lst}(i)) \mid i \text{ in } \{1..\text{len lst} \} \cdot P(\text{lst}(i)) \}
\]

A.3.4 Map Enumerations

Let \( a \) and \( b \) range over values of type \( A \) and \( B \), respectively (allowing indexing for solving ambiguity). Then the below expressions are simple map enumerations:

\[
\text{type} \\
A, B \\
M = A \rightarrow m B \\
\text{value} \\
a,a_1,a_2,...,a_3:A, b,b_1,b_2,...,b_3:B \\
[ ], [a \rightarrow b], ..., [a_1 \rightarrow b_1,a_2 \rightarrow b_2,...,a_3 \rightarrow b_3] \forall \in M
\]
The last line below expresses map comprehension:

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>type</strong></td>
</tr>
<tr>
<td>A, B, C, D</td>
</tr>
<tr>
<td>M = A \rightarrow B</td>
</tr>
<tr>
<td>F = A \rightarrow C</td>
</tr>
<tr>
<td>G = B \rightarrow D</td>
</tr>
<tr>
<td>P = A \rightarrow Bool</td>
</tr>
<tr>
<td><strong>value</strong></td>
</tr>
<tr>
<td>comprehend: MxFxGxP \rightarrow (C \rightarrow D)</td>
</tr>
<tr>
<td>comprehend(m,F,G,P) =</td>
</tr>
<tr>
<td>[ F(a) \rightarrow G(m(a)) \mid a:A \land a \in \text{dom} m \land P(a) ]</td>
</tr>
</tbody>
</table>

**A.3.5 Set Operations**

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>value</strong></td>
</tr>
<tr>
<td>( \in): A \times A-infset \rightarrow Bool</td>
</tr>
<tr>
<td>( \not\in): A \times A-infset \rightarrow Bool</td>
</tr>
<tr>
<td>( \cup): A-infset \times A-infset \rightarrow A-infset</td>
</tr>
<tr>
<td>( \cup): (A-infset)-infset \rightarrow A-infset</td>
</tr>
<tr>
<td>( \cap): A-infset \times A-infset \rightarrow A-infset</td>
</tr>
<tr>
<td>( \cap): (A-infset)-infset \rightarrow A-infset</td>
</tr>
<tr>
<td>( \setminus): A-infset \times A-infset \rightarrow A-infset</td>
</tr>
<tr>
<td>( \subset): A-infset \times A-infset \rightarrow Bool</td>
</tr>
<tr>
<td>( \subseteq): A-infset \times A-infset \rightarrow Bool</td>
</tr>
<tr>
<td>( \neq): A-infset \times A-infset \rightarrow Bool</td>
</tr>
<tr>
<td>card: A-infset \rightarrow Nat</td>
</tr>
<tr>
<td><strong>examples</strong></td>
</tr>
<tr>
<td>a \in {a,b,c}</td>
</tr>
<tr>
<td>a \not\in {}, a \not\in {b,c}</td>
</tr>
<tr>
<td>{a,b,c} \cup {a,b,d,e} = {a,b,c,d,e}</td>
</tr>
<tr>
<td>\cup{{a},{a,b},{a,d}} = {a,b,d}</td>
</tr>
<tr>
<td>{a,b,c} \cap {c,d,e} = {c}</td>
</tr>
<tr>
<td>\cap{{a},{a,b},{a,d}} = {a}</td>
</tr>
<tr>
<td>{a,b,c} \setminus {c,d} = {a,b}</td>
</tr>
<tr>
<td>{a,b} \subset {a,b,c}</td>
</tr>
<tr>
<td>{a,b,c} \subseteq {a,b,c}</td>
</tr>
</tbody>
</table>
\[
\{a,b,c\} = \{a,b,c\} \\
\{a,b,c\} \neq \{a,b\} \\
\text{card} \{\} = 0, \text{card} \{a,b,c\} = 3
\]

Annotations:

- \(\in\) The membership operator expresses that an element is member of a set.
- \(\notin\) The nonmembership operator expresses that an element is not member of a set.
- \(\cup\) The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
- \(\cap\) The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
- \(\setminus\) The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
- \(\subseteq\) The proper subset operator expresses that all members of the left operand set are also in the right operand set.
- \(\subset\) The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
- \(\equiv\) The equal operator expresses that the two operand sets are identical.
- \(\neq\) The nonequal operator expresses that the two operand sets are not identical.
- \(\text{card}\) The cardinality operator gives the number of elements in a (finite) set.

The operations can be defined as follows:

\[
\begin{align*}
\text{value} & \\
\text{let } a:\text{A} & \cdot a \in s' \lor a \in s'' \\
\text{let } a:\text{A} & \cdot a \in s' \land a \in s'' \\
\text{let } a:\text{A} & \cdot a \in s' \land a \notin s'' \\
\text{let } a:\text{A} & \cdot a \in s'' \land s' \subset s'' \land s \subseteq s' \land s'' \subseteq s \\
\text{let } a:\text{A} & \cdot a \in s' \land a \in s' \lor a \notin s'' \\
\text{card } s & \equiv \begin{cases} 
\text{if } s = \{\} & \text{then } 0 \\
\text{let } a:\text{A} & \cdot a \in s \text{ in } 1 + \text{card} \{s \setminus \{a\}\} \text{ end end} \\
\text{pre } s/∗\text{ is a finite set }*/ \\
\text{card } s & \equiv \text{chaos }/*\text{ tests for infinity of } s/∗
\end{cases}
\end{align*}
\]
A.3.6 Cartesian Operations

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
</tr>
<tr>
<td>A, B, C</td>
</tr>
<tr>
<td>g0: G0 = A × B × C</td>
</tr>
<tr>
<td>g1: G1 = ( A × B × C )</td>
</tr>
<tr>
<td>g2: G2 = ( A × B ) × C</td>
</tr>
<tr>
<td>g3: G3 = A × ( B × C )</td>
</tr>
</tbody>
</table>

| value |
| va:A, vb:B, vc:C, vd:D |
| (va,vb,vc):G0, |
| (va,vb,vc):G1 |
| ((va,vb),vc):G2 |
| (va3,(vb3,vc3)):G3 |

| decomposition expressions |
| let (a1,b1,c1) = g0, |
| (a1′,b1′,c1′) = g1 in .. end |
| let ((a2,b2),c2) = g2 in .. end |
| let (a3,(b3,c3)) = g3 in .. end |

A.3.7 List Operations

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
</tr>
<tr>
<td>hd: A^ω ⊢ A</td>
</tr>
<tr>
<td>tl: A^ω ⊢ A^ω</td>
</tr>
<tr>
<td>len: A^ω ⊢ Nat</td>
</tr>
<tr>
<td>inds: A^ω → Nat-infset</td>
</tr>
<tr>
<td>elems: A^ω → A-infset</td>
</tr>
<tr>
<td>.(,): A^ω × Nat ⊢ A</td>
</tr>
<tr>
<td>⊖: A^* × A^ω → A^ω</td>
</tr>
<tr>
<td>=: A^ω × A^ω → Bool</td>
</tr>
<tr>
<td>≠: A^ω × A^ω → Bool</td>
</tr>
</tbody>
</table>

| examples |
| hd⟨a1,a2,...,am⟩=a1 |
| tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩ |
| len⟨a1,a2,...,am⟩=m |
| inds⟨a1,a2,...,am⟩={1,2,...,m} |
| elems⟨a1,a2,...,am⟩={a1,a2,...,am} |
\[ \langle a_1, a_2, \ldots, a_m \rangle(i) = a_i \]
\[ \langle a, b, c \rangle \hat{\langle} \langle a, b, d \rangle = \langle a, b, c, a, b, d \rangle \]
\[ \langle a, b, c \rangle = \langle a, b, c \rangle \]
\[ \langle a, b, c \rangle \neq \langle a, b, d \rangle \]

**Annotations:**

- **hd** Head gives the first element in a nonempty list.
- **tl** Tail gives the remaining list of a nonempty list when Head is removed.
- **len** Length gives the number of elements in a finite list.
- **inds** Indices gives the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems** Elements gives the possibly infinite set of all distinct elements in a list.
- **\( \ell(i) \)** Indexing with a natural number, \( i \) larger than 0, into a list \( \ell \) having a number of elements larger than or equal to \( i \), gives the \( i \)th element of the list.
- **\( \hat{\langle} \)** Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- **=** The equal operator expresses that the two operand lists are identical.
- **\( \neq \)** The nonequal operator expresses that the two operand lists are not identical.

The operations can also be defined as follows:

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>value</strong></td>
</tr>
<tr>
<td>is_finite_list: ( A^\omega \rightarrow \mathbf{Bool} )</td>
</tr>
<tr>
<td><strong>len</strong> ( q ) ( \equiv )</td>
</tr>
<tr>
<td>case is_finite_list( (q) ) of</td>
</tr>
<tr>
<td>true ( \rightarrow ) if ( q = \langle \rangle ) then 0 else 1 + len tl ( q ) end,</td>
</tr>
<tr>
<td>false ( \rightarrow ) chaos end</td>
</tr>
<tr>
<td><strong>inds</strong> ( q ) ( \equiv )</td>
</tr>
<tr>
<td>case is_finite_list( (q) ) of</td>
</tr>
<tr>
<td>true ( \rightarrow ) { i</td>
</tr>
<tr>
<td>false ( \rightarrow ) { i</td>
</tr>
<tr>
<td><strong>elems</strong> ( q ) ( \equiv ) { q(i)</td>
</tr>
<tr>
<td>( q(i) ) ( \equiv )</td>
</tr>
<tr>
<td>if ( i=1 )</td>
</tr>
</tbody>
</table>
then
  if \( q \neq \langle \rangle \)
    then let \( a:A, q':Q \cdot q = \langle a \rangle \hat{q}' \) in a end
  else chaos end
else \( q(i-1) \) end

\( f_q \hat{q} \equiv \langle \hat{f}_q \rangle \)

\( i_q \equiv \langle \hat{i}_q \rangle \)

\( \hat{\text{inds}} \cdot \hat{i}_q \equiv \hat{\text{inds}} \cdot \hat{i}_q \wedge \forall i: \text{Nat} \cdot i \in \text{inds} \rightarrow i_q'(i) = i_q''(i) \)

\( i_q' \neq i_q'' \equiv \sim (i_q' = i_q'') \)

### A.3.8 Map Operations

#### Formal Expressions

**value**

\( m(a): M \rightarrow A \sim B, m(a) = b \)

**dom**

\( \text{dom} [a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots, a_n \mapsto b_n] = \{a_1, a_2, \ldots, a_n\} \)

**rng**

\( \text{rng} [a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots, a_n \mapsto b_n] = \{b_1, b_2, \ldots, b_n\} \)

\( \hat{\cdot}: M \times M \rightarrow M \) [override extension]

\[ [a \mapsto b, a' \mapsto b', a'' \mapsto b''] \hat{\cdot} [a' \mapsto b'', a'' \mapsto b] = [a \mapsto b, a' \mapsto b'', a'' \mapsto b] \]

\( \cup: M \times M \rightarrow M \) [merge \( \cup \)]

\[ [a \mapsto b, a' \mapsto b', a'' \mapsto b''] \cup [a''' \mapsto b'''] = [a \mapsto b, a' \mapsto b', a'' \mapsto b''', a''' \mapsto b'''] \]

\( \setminus: M \times A-\text{infset} \rightarrow M \) [restriction by]

\[ [a \mapsto b, a' \mapsto b', a'' \mapsto b'] \setminus \{a\} = [a' \mapsto b', a'' \mapsto b'] \]

\( /: M \times A-\text{infset} \rightarrow M \) [restriction to]

\[ [a \mapsto b, a' \mapsto b', a'' \mapsto b'''] / \{a', a''\} = [a' \mapsto b', a'' \mapsto b'''] \]

\( =, \neq: M \times M \rightarrow \text{Bool} \)

\( \circ: (A \ _{\text{mr}} \ B) \times (B \ _{\text{mr}} \ C) \rightarrow (A \ _{\text{mr}} \ C) \) [composition]
\[ [a\mapsto b, a'\mapsto b'] \circ [b\mapsto c, b'\mapsto c', b''\mapsto c''] = [a\mapsto c, a'\mapsto c'] \]

Annotations:

- \( m(a) \) Application gives the element of which \( a \) maps to in the map \( m \).
- \( \text{dom} \) Domain/definition set gives the set of values which maps to in a map.
- \( \text{rng} \): Range/image set gives the set of values which are mapped to in a map.
- \( \dagger \) Override/extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some “pairings” of the right operand map.
- \( \cup \) Merge. When applied to two operand maps, it gives a merge of these maps.
- \( \setminus \) Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- \( / \) Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- \( = \) The equal operator expresses that the two operand maps are identical.
- \( \neq \) The nonequal operator expresses that the two operand maps are not identical.
- \( \circ \) Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, \( m_1 \), to the range elements of the right operand map, \( m_2 \), such that if \( a \) is in the definition set of \( m_1 \) and maps into \( b \), and if \( b \) is in the definition set of \( m_2 \) and maps into \( c \), then \( a \), in the composition, maps into \( c \).

The map operations can also be defined as follows:

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{rng} \ m \equiv { m(a) \mid a:A \cdot a \in \text{dom} \ m } )</td>
</tr>
<tr>
<td>( m_1 \dagger m_2 \equiv \begin{cases} a\mapsto b \mid a:A, b:B \cdot a \in \text{dom} \ m_1 \setminus \text{dom} \ m_2 \land b=m_1(a) \lor a \in \text{dom} \ m_2 \land b=m_2(a) \end{cases} )</td>
</tr>
<tr>
<td>( m_1 \cup m_2 \equiv \begin{cases} a\mapsto b \mid a:A, b:B \cdot a \in \text{dom} \ m_1 \land b=m_1(a) \lor a \in \text{dom} \ m_2 \land b=m_2(a) \end{cases} )</td>
</tr>
<tr>
<td>( m \setminus s \equiv \begin{cases} a\mapsto m(a) \mid a:A \cdot a \in \text{dom} \ m \setminus s \end{cases} )</td>
</tr>
<tr>
<td>( m / s \equiv \begin{cases} a\mapsto m(a) \mid a:A \cdot a \in \text{dom} \ m \cap s \end{cases} )</td>
</tr>
<tr>
<td>( m_1 = m_2 \equiv \text{dom} \ m_1 = \text{dom} \ m_2 \land \forall a:A \cdot a \in \text{dom} \ m_1 \Rightarrow m_1(a) = m_2(a) )</td>
</tr>
</tbody>
</table>
\[ m_1 \neq m_2 \equiv \neg (m_1 = m_2) \]
\[ m^n \equiv \{ a \mapsto c \mid a : A, c : C, a \in \text{dom } m \land c = n(m(a)) \} \]
\[ \text{pre } \text{rng } m \subseteq \text{dom } n \]

### A.4 \(\lambda\)-Calculus and Functions

RSL supports function expressions for \(\lambda\)-abstraction.

#### A.4.1 The \(\lambda\)-Calculus Syntax

**Formal Expressions**

```plaintext
<table>
<thead>
<tr>
<th>type</th>
<th>/* A BNF Syntax: */</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L)</td>
<td>::= (V)</td>
</tr>
<tr>
<td>(V)</td>
<td>::= /* variables, i.e. identifiers */</td>
</tr>
<tr>
<td>(F)</td>
<td>::= (\lambda)(V) \cdot (L)</td>
</tr>
<tr>
<td>(A)</td>
<td>::= ( (L)/(L) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>/* Examples */</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L):</td>
<td>e, f, a, ...</td>
</tr>
<tr>
<td>(V):</td>
<td>x, ...</td>
</tr>
<tr>
<td>(F):</td>
<td>(\lambda) x \cdot e, ...</td>
</tr>
<tr>
<td>(A):</td>
<td>f a, (f a), f(a), (f)(a), ...</td>
</tr>
</tbody>
</table>
```

#### A.4.2 Free and Bound Variables

**Formal Expressions**

Let \(x, y\) be variable names and \(e, f\) be \(\lambda\)-expressions.

- (V): Variable \(x\) is free in \(x\).
- (F): \(x\) is free in \(\lambda y \cdot e\) if \(x \neq y\) and \(x\) is free in \(e\).
- (A): \(x\) is free in \(f(e)\) if it is free in either \(f\) or \(e\) (i.e., also in both).

#### A.4.3 Substitution

In RSL, the following rules for substitution apply:

**Formal Expressions**

- \(\text{subst}(N/x|x) \equiv N;\)
A.4.4 α-Renaming and β-Reduction

Formal Expressions

- **α-renaming**: $\lambda x \cdot M$
  
  If $x$ and $y$ are distinct variables then replacing $x$ by $y$ in $\lambda x \cdot M$ results in $\lambda y \cdot \text{subst}(y/x|M)$:
  
  We can rename the formal parameter of a $\lambda$-function expression provided that no free variables of its body $M$ thereby become bound.

- **β-reduction**: $(\lambda x \cdot M)(N)$
  
  All free occurrences of $x$ in $M$ are replaced by the expression $N$ provided that no free variables of $N$ thereby become bound in the result.
  
  $(\lambda x \cdot M)(N) \equiv \text{subst}(N/x|M)$

A.4.5 Function Signatures

For some functions, we want to abstract from the function body:

Formal Expressions

**value**

- $\text{obs}_{\text{Pos}_{\text{Aircraft}}}: \text{Aircraft} \to \text{Pos},$
- $\text{move}: \text{Aircraft} \times \text{Dir} \to \text{Aircraft},$
A.4.6 Function Definitions

Functions — with body — can be defined explicitly

**Formal Expressions**

\[
\begin{align*}
\text{value} \quad f & : A \times B \times C \to D \\
f(a,b,c) & \equiv \text{Value}_\text{Expr} \\

g & : B\text{-infset} \times (D \to C\text{-set}) \sim \to A^* \\
g(bs,dm) & \equiv \text{Value}_\text{Expr} \\
\text{pre} & \; \mathcal{P}(dm) \\
\text{post} & \; \mathcal{P}_1(d) \\
\text{post} & \; \mathcal{P}_3(al) \\
\end{align*}
\]

or implicitly

**Formal Expressions**

\[
\begin{align*}
\text{value} \quad f & : A \times B \times C \to D \\
f(a,b,c) & \equiv d \\
\text{post} & \; \mathcal{P}_1(d) \\

g & : B\text{-infset} \times (D \to C\text{-set}) \sim \to A^* \\
g(bs,dm) & \equiv \text{al} \\
\text{pre} & \; \mathcal{P}_2(dm) \\
\text{post} & \; \mathcal{P}_3(al) \\
\end{align*}
\]

The symbol \( \sim \) indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

A.5 Further Applicative Expressions

A.5.1 Let Expressions

Simple (i.e., nonrecursive) let expressions:

**Formal Expressions**

\[
\text{let} \; a = \xi_d \; \text{in} \; \xi_d(a) \; \text{end}
\]

is an “expanded” form of
Recursive \textbf{let} expressions are written as:

\[
(\lambda a.E_b(a))(E_d)
\]

Predicative \textbf{let} expressions:

\[
\text{let } a:A \cdot P(a) \text{ in } B(a) \text{ end}
\]

express the selection of a value \(a\) of type \(A\) which satisfies a predicate \(P(a)\) for evaluation in the body \(B(a)\).

\textit{Patterns} and \textit{wild cards} can be used:

\[
\begin{align*}
\text{let } \{a\} \cup s &= \text{set in ... end} \\
\text{let } \{a,\_\} \cup s &= \text{set in ... end} \\
\text{let } (a,b,\ldots,c) &= \text{cart in ... end} \\
\text{let } (a,\_,\ldots,c) &= \text{cart in ... end} \\
\text{let } (a)^{\ell} &= \text{list in ... end} \\
\text{let } (a,b)^{\ell} &= \text{list in ... end} \\
\text{let } [a \mapsto b] \cup m &= \text{map in ... end} \\
\text{let } [a \mapsto b,\_] \cup m &= \text{map in ... end}
\end{align*}
\]
A.5.2 Conditionals

Various kinds of conditional expressions are offered by RSL:

### Formal Expressions

\[
\begin{align*}
\text{if } & b_{\text{expr}} \text{ then } c_{\text{expr}} \text{ else } a_{\text{expr}} \text{ end} \\
\text{if } & b_{\text{expr}} \text{ then } c_{\text{expr}} \text{ end } \equiv */\text{ same as: } */ \\
\text{if } & b_{\text{expr}} \text{ then } c_{\text{expr}} \text{ end} \\
\text{elsif } & b_{\text{expr}_1} \text{ then } c_{\text{expr}_1} \\
\text{elsif } & b_{\text{expr}_2} \text{ then } c_{\text{expr}_2} \\
\text{elsif } & b_{\text{expr}_3} \text{ then } c_{\text{expr}_3} \\
\text{... elsif } & b_{\text{expr}_n} \text{ then } c_{\text{expr}_n} \text{ end} \\
\text{case } & \text{expr of} \\
& \text{choice_pattern}_1 \rightarrow \text{expr}_1, \\
& \text{choice_pattern}_2 \rightarrow \text{expr}_2, \\
& \text{...} \\
& \text{choice_pattern}_n \text{ or } \text{wild_card} \rightarrow \text{expr}_n \\
\text{end}
\end{align*}
\]

A.5.3 Operator/Operand Expressions

### Formal Expressions

\[
\begin{align*}
\langle \text{Expr} \rangle := \\
\langle \text{Prefix(Op)} \rangle \langle \text{Expr} \rangle \\
\mid \langle \text{Expr} \rangle \langle \text{Infix(Op)} \rangle \langle \text{Expr} \rangle \\
\mid \langle \text{Expr} \rangle \langle \text{Suffix(Op)} \rangle \\
\mid ... \\
\langle \text{Prefix(Op)} \rangle := \\
- | \sim | \cup | \cap | \text{card} | \text{len} | \text{inds} | \text{elems} | \text{hd} | \text{tl} | \text{dom} | \text{rng} \\
\langle \text{Infix(Op)} \rangle := \\
= | \neq | + | - | * | \uparrow | / | < | \leq | > \Rightarrow \\
| \in | \notin | \cup | \cap | \setminus | \subseteq | \supseteq | \bowtie | \circ \\
\langle \text{Suffix(Op)} \rangle := !
\end{align*}
\]

A.6 Imperative Constructs

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative con-
structs. Imperative constructs are thus inevitable in RSL.

### A.6.1 Variables and Assignment

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. <strong>variable</strong> v:Type := expression</td>
</tr>
<tr>
<td>1. v := expr</td>
</tr>
</tbody>
</table>

### A.6.2 Statement Sequences and skip

Sequencing is done using the “;” operator. **skip** is the empty statement having no value or side effect.

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. <strong>skip</strong></td>
</tr>
<tr>
<td>3. stm_1;stm_2;...;stm_n</td>
</tr>
</tbody>
</table>

### A.6.3 Imperative Conditionals

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. <strong>if</strong> expr <strong>then</strong> stm_c <strong>else</strong> stm_a <strong>end</strong></td>
</tr>
<tr>
<td>5. <strong>case</strong> e of: p_1→S_1(p_1),...,p_n→S_n(p_n) <strong>end</strong></td>
</tr>
</tbody>
</table>

### A.6.4 Iterative Conditionals

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. <strong>while</strong> expr <strong>do</strong> stm <strong>end</strong></td>
</tr>
<tr>
<td>7. <strong>do</strong> stmt <strong>until</strong> expr <strong>end</strong></td>
</tr>
</tbody>
</table>

### A.6.5 Iterative Sequencing

<table>
<thead>
<tr>
<th>Formal Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>8. <strong>for</strong> b <strong>in</strong> list,expr • P(b) <strong>do</strong> S(b) <strong>end</strong></td>
</tr>
</tbody>
</table>
A.7 Process Constructs

A.7.1 Process Channels

Let A, B stand for types of channel messages and KIdx stand for channel array indexes. Then

\[
\text{Formal Expressions}
\]

\[
\begin{align*}
\text{channel } & \text{c:A} \\
\text{channel } & \{ \text{k}[i]:B \cdot i:KIdx \}
\end{align*}
\]

declare a channel, c, and an array of channels, k, whose individual channels, k[i], are able to communicate values of the designated types.

A.7.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q(i) stand for process expressions\(^2\) then

\[
\text{Formal Expressions}
\]

\[
\begin{align*}
P() & \parallel Q(i) \quad \text{Parallel composition} \\
P() & [ Q(i) \quad \text{Nondeterministic External Choice (either/or)} \\
P() & [ [ Q(i) \quad \text{Nondeterministic Internal Choice (either/or)} \\
P() & || Q() \quad \text{Interlock Parallel composition}
\end{align*}
\]

expresses the parallel (||) of two processes, the nondeterministic choice between two processes, either external ([]) or internal ([[]). The interlock (||) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

A.7.3 Input/Output Events

Let c and k[i] designate channels of type A, and let e designate an expression also of type A. Then

\[
\text{Formal Expressions}
\]

\[
\begin{align*}
c & ?, k[i] ? \quad \text{Input expression (a clause)} \\
c & ! e, k[i] ! e \quad \text{Output clause (a statement)}
\end{align*}
\]

expresses the willingness to engage in an event that “reads” an input, and respectively “writes” an output.

\(^2\)Both expressions (P() and (Q(i)) name process definitions (P respectively Q). P has no formal parameters. Q has, as only parameter, a channel array index. The former, P(), thus invokes P with no arguments and the latter, Q(i), invokes Q with a channel array index argument.
A.7.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

\[
\begin{align*}
\text{value} \\
\text{P: Unit} & \rightarrow \text{in c out k[i] Unit} \\
\text{Q: i:KIdx} & \rightarrow \text{out c in k[i] Unit} \\
\text{P()} & \equiv ... \text{c ? ... k[i] } ! \text{ e ...} \\
\text{Q(i)} & \equiv ... \text{k[i] ? ... e } ! \text{ e ...}
\end{align*}
\]

The process function definitions (i.e., their bodies) express possible events.

A.8 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, one or more values (including functions), zero, one or more variables, zero, one or more channels and zero, one or more axioms listed under respective type, variable, channel, value and axiom “headers”. We prefer to list their order as shown:

\[
\begin{align*}
\text{type} \\
\text{...} \\
\text{variable} \\
\text{...} \\
\text{channel} \\
\text{...} \\
\text{value} \\
\text{...} \\
\text{axiom} \\
\text{...}
\end{align*}
\]

In practice a full specification repeats the above listings many times, once for each “module” (i.e., aspect, facet, view) of specification. Each of these modules may be “wrapped” into scheme, class or object definitions.\(^3\)

---

\(^3\)For schemes, classes and objects we refer to Vol. 2, Chap. 10 of this series of textbooks.
Appendix B

CafeOBJ: The Language

Someone to provide a primer of CafeOBJ: Syntax, Semantics and Proof System