APM

POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE CB3 0RD UNITED KINGDOM
+44 1223 515010 • Fax +44 1223 359779 • Email: apm@ansa.co.uk • URL: http://www.ansa.co.uk

Configuration Semantics for FlexiNet Applications

W. T. Harwood

Abstract

Adapters are protocol transformers. This document defines a simple rule based framework for the description of adapters within FlexiNet. The framework defines notions of adapter type, adapter cost function and adapter context constraint and the rules for adapter composition and for the conformance of an adapter to set of constraints. These adapter descriptions and inference rules are the basis for a framework of “protocol negotiation”.

Doc. No. XXX	**Draft** ANSA Report	3 December 1997

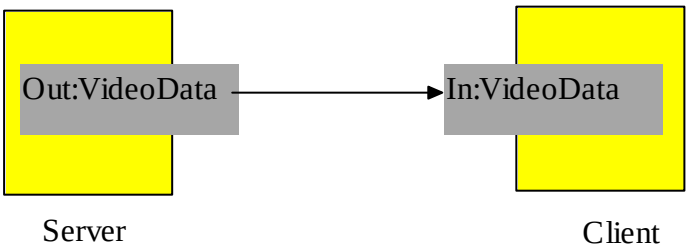
Distribution:
Supersedes:
Superseded by:

Table of Contents

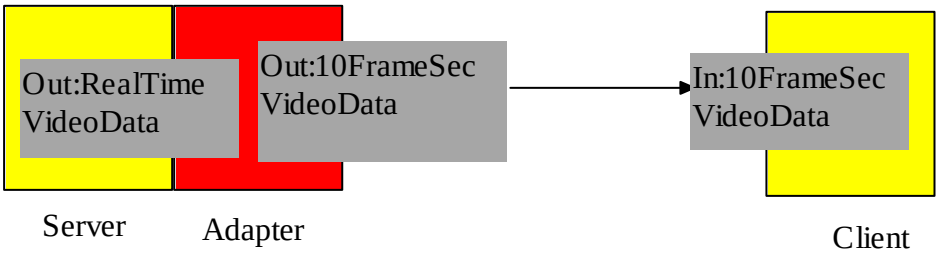
1 IntroductioN

1.1 The Problem

FlexiNet offers a component based approach to defining the interaction policies between processes. The situation is illustrated by considering a small example in which a client wishes to connect to a video service provided by a server. The simplest situation is where the client requests a simple video interface and the server offers an identical interface. The interface between client and server can be thought of as an *out* channel on the server and an *in* channel on the client and both channels have identical data types, say, VideoData. The server and client descriptions might look like:-



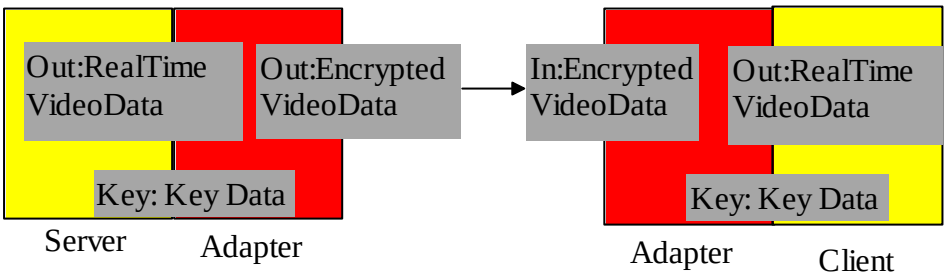
This simple situation may be made more complicated by there being a mismatch between the data types of the channels on the Server and Client. For example the server might provide a channel with RealTimeVideoData and the client may only accept a data type supporting a limited capacity, such as, 10FramesSecVideoData. The resolution of the type mismatch is resolved by placing adapters on the client side, the server side or both, thus, in effect, creating a new server, a new client or both. In the case of the server deploying an adapter the original video service is hidden from the client and the new service offered to the client:-



Adapters are type transforming processes. In their simplest form they are transformer *functions*, however in the general case they are processes that transform

protocols. In order to fit services and adapters together we need an approximate way of describing the behaviour of each. We call this approximate description the *process type*. The process type defines what channels the process is willing to communicate over and what data types it wishes to pass over these channels.

Generally a process interface has mandatory components and optional components that can be used by an adapter to achieve its translation. The most obvious example of this is use of key information in encryption adapters. A service is willing to offer video data and can be adapted to offer encrypted video data. The encryption adapter requires a key as input. The encryption adapter requires a channel to the process infrastructure to obtain the key. We will call such additional interfaces "helper interfaces".



Processes interact by their willingness to transmit and receive typed data over named channels. We can characterise processes by "process types" that give a coarse grained view of the behaviour of processes. From a suitably distant perspective all process of the same type "behave the same".

1.2 Choosing Adapters

Adapters are objects that may be classified in various ways. They may be classified with respect to:-

- the transformation they carry out
- the static context in which they may be used
- the "adapter context" in which they may used
- the fixed cost of their deployment
- the variable cost of their deployment

We call the various ways in which adapters may be classified perspectives. In general the selection of an adapter is constrained in each perspective. An adapter is selected for the transformation it performs, for the context in which it may be used and for the cost of its deployment.

1.3 Approaches to Adapter Selection

There seem to be three basic alternatives to selecting adapters. The first possibility exists when there are only a small number of sensible adapters for any given interaction. In this case a protocol very similar to the SSL handshake protocol is all that is necessary. A client can fill in a small number of parameters to describe the options that are acceptable and the server responds to those options if possible. Allowing alternatives in this structure is straightforward in that rather than providing a single “option list” the client provides a set options lists to the server (possibly in priority order)..

The second approach is applicable when there are many possible combinations of basic adapters that can be used to achieve an overall adapter type but there is a simple global notion of “best” for all participants. In such cases it is possible to pre-compute the best solutions for each type and store these. Negotiation is then a matter of asking for the best adapter of a given type.

Finally the third approach is applicable when there are a large number of possible adapters and no simple global notion of “best” adapter for a type. In such cases adapters need to be built on an ad-hoc basis.

In each case there is a choice of where to place adapters, at the client end, the server end or both. Both client side and server side have requirements for the adapters in terms of functionality of the adapters, the cost of use of the adapters and contextual constraints. Moreover, even though we speak of client and server roles there may be many clients or servers involved in the interaction.

Our view of negotiation is that a client process requests a link to a server process. Both client and server specify the adapters that they are willing to provide. Negotiation is the process of selecting adapters that realise the end-to-end transport requirement by composing client and server adapters whilst meeting any additional cost and/or context constraint.

1.4 The Rest of the Document

The rest of the document creates a framework for discussing the selection of adapters. It is primarily concerned with laying down a framework within which adapter types and constraints can be expressed and adapter compliance to constraints can be assessed.

2 Constraints

2.1 A Formalism for Perspectives

A common problem in producing a “specification” formalism is to attempt to create a uniform notation that carries a very high work load. Rather than produce a single logical system that integrates the disparate aspect of adapter description we define an approach to combining logical systems and then give a separate logical system for each aspect of adapter description.

When an adapter **A** may be classified from some perspective **p** by a description **D** we write this as $\mathbf{A} \triangleright_{\mathbf{p}} \mathbf{D}$.

Given an adapter **A** and a collection of classifications along different perspectives $\mathbf{A} \triangleright_{1} \mathbf{D_1}$, $\mathbf{A} \triangleright_{2} \mathbf{D_2}$,... , $\mathbf{A} \triangleright_{\mathbf{n}} \mathbf{D_n}$ then the classifications can be combined to form the product classification $\mathbf{A} \triangleright_{(1,2,\ldots,n)} (\mathbf{D_1}, \mathbf{D_2}, \ldots, \mathbf{D_n})$. That is we adopt the rule:-

$$\frac{\mathbf{A} \triangleright_1 \mathbf{D}_1 \quad \mathbf{A} \triangleright_2 \mathbf{D}_2 \quad \cdots \quad \mathbf{A} \triangleright_n \mathbf{D}_n}{\mathbf{A} \triangleright_{1,2,\ldots,n} (\mathbf{D}_1, \mathbf{D}_2, \cdots, \mathbf{D}_n)}$$

For each perspective of classification we define a language for writing descriptions of adapters from that perspective.

Adapters may also be composed in two ways to form new adapters. Adapters may be sequenced written **A;B** for adapters **A** and **B**, and adapters may be combined in parallel written **A** || **B** . For each perspective of classification inference rules define how classifications are combined to create the classification of compound adapters.

2.2 Notation

Throughout each subsection below we introduce a particular form of constraint and give the rules for combining constraints of each type.

Within a subsection the constraint is always called " $\triangleright$ " without a subscript whenever this is possible without creating ambiguity.

Adapters are viewed as transforming the interface of a process on the left hand side of the adapter to a transport interface on the right hand side of an adapter.

2.3 Static Context

The static context perspective of adapters describes fixed attributes of an adapter. For example certain adapters may only be used in certain location or by certain machines. Any compound adapter simply inherits the static context constraints.

An atomic static context constraint is simply a propositional symbol or token that is chosen to represent a property. Atomic propositions can be combined to form boolean expressions. Each adapter is described by a set static context axioms. A compound adapter satisfies a static context constraint if each of its component parts satisfies the constraint. In order to minimise the number of static context assertions required for adapters we wish to treat atomic propositions not mentioned in the axioms of an adapter as don't care terms for that adapter. In order to do this we introduce the notion of the set of atomic propositions associated with an adapter, which for an adapter **A** will be referred to as Atoms(**A**). Any atomic proposition not in mentioned in Atoms(**A**) will be regarded as a don't care term for **A**. A don't care term can be assumed true or false provided that this is done consistently throughout. In order to manage this consistency some book keeping mechanism is required. The constraint $\triangleright$ is annotated with a set of assumptions that indicate, for an adapter **A** what decisions have been made about atomic propositions not in Atoms(**A**). So $\mathbf{A} \triangleright_{\sigma} \varphi$ means that φ holds for **A** under the set of assumptions σ. The assumptions in the assumption set are of the form (**A**, φ, **true**) and (**A**, φ, **false**) meaning, respectively, φ is assumed true in **A** and φ is assumed false in **A**. A set of assumptions is consistent if a variable is not assumed both true and false for the same adapter and the variables with assumed values are not in the atoms of the adapter for which the value is assumed.

$$\frac{\mathbf{A} \triangleright_{s} j \quad \mathbf{B} \triangleright_{s'} j}{\mathbf{A}; \mathbf{B} \triangleright_{s \cup s'} j} \text{ provided } \sigma \cup \sigma' \text{ is consistent.}$$

$$\frac{\mathbf{A} \triangleright_{s} j \quad \mathbf{B} \triangleright_{s'} j}{\mathbf{A} || \mathbf{B} \triangleright_{s \cup s'} j} \text{ provided } \sigma \cup \sigma' \text{ is consistent}$$

$$\frac{j \notin \mathit{Atoms}(\mathbf{A}) \quad (\mathbf{A}, j, \mathbf{true}) \in s}{\mathbf{A} \triangleright_{s} j} \qquad \frac{j \notin \mathit{Atoms}(\mathbf{A}) \quad (\mathbf{A}, j, \mathbf{false}) \in s}{\mathbf{A} \triangleright_{s} \neg j}$$

provided σ is consistent and φ is an atomic proposition.

Constraints may be combined by boolean operations. Throughout the following rules $\sigma \cup \sigma'$ is required to be consistent.

$$\frac{\mathbf{A} \rhd_s j \wedge y}{\mathbf{A} \rhd_s j} \qquad \frac{\mathbf{A} \rhd_s j \wedge y}{\mathbf{A} \rhd_s j} \qquad \frac{\mathbf{A} \rhd_s j \quad \mathbf{A} \rhd_{s'} y}{\mathbf{A} \rhd_{s \cup s'} j \wedge y}$$

$$\frac{\mathbf{A} \rhd_s \neg j \quad \mathbf{A} \rhd_{s'} j \vee y}{\mathbf{A} \rhd_{s \cup s'} y} \qquad \frac{\mathbf{A} \rhd_s \neg y \quad \mathbf{A} \rhd_{s'} j \vee y}{\mathbf{A} \rhd_{s \cup s'} y} \qquad \frac{\mathbf{A} \rhd_s j}{\mathbf{A} \rhd_s j \vee y} \qquad \frac{\mathbf{A} \rhd_s y}{\mathbf{A} \rhd_s j \vee y}$$

$$\frac{\mathbf{A} \rhd_s j \Rightarrow y \quad \mathbf{A} \rhd_{s'} j}{\mathbf{A} \rhd_{s \cup s'} y} \qquad \frac{\begin{array}{c}\overline{\mathbf{A} \rhd_s j} \\ \vdots \\ \mathbf{A} \rhd_s y\end{array}}{\mathbf{A} \rhd_s j \Rightarrow y}$$

$$\frac{\mathbf{A} \rhd_s j \quad \mathbf{A} \rhd_{s'} \neg j}{\mathbf{A} \rhd_{s \cup s'} \bot} \qquad \frac{\begin{array}{c}\overline{\mathbf{A} \rhd_s j} \\ \vdots \\ \mathbf{A} \rhd_s \bot\end{array}}{\mathbf{A} \rhd_s \neg j}$$

2.4 Adapter Context Constraints

Adapter context constraints record the history of adapters. Any adapters may require certain context constraints to be met on its left and may propagate context assertions to its right. Thus the adapter perspective that we use is a pair of sets (α, β) where α is the set of requirements for the adapter and β is the set of conditions satisfied by the adapter. Requirement conditions are propositional formula defined in terms of some set of propositional letters. The elements of α are propositional formula, the elements β of are atomic propositions which are either propositional letters or negated propositional letters.

$\mathbf{A} \rhd (\alpha, \beta)$ means that α is the complete set of **A** requirements to the left and β is a complete set of atomic historical propositions that **A** obeys. The rules for combining satisfaction sets are:-

$$\frac{\mathbf{A} \rhd (a_A, b_A) \quad \mathbf{B} \rhd (a_B, b_B) \quad b_A \text{`}_{a \in a_B} a}{\mathbf{A};\mathbf{B} \rhd (a_A, b_A \cup b_B)}$$

where $b \text{`}_{a \in a_B} a$ means that β_A entails every α in the set α_B by the rules of propositional logic.

$$\frac{\mathbf{A} \triangleright (a_A, b_A) \quad \mathbf{B} \triangleright (a_B, b_B)}{\mathbf{A} \| \mathbf{B} \triangleright (a_A \cup a_B, b_A \cup b_B)}$$

Aside: The static context model of assumptions is not used here.

2.5 Fixed Costs

Fixed costs are additive. Let **x** and **y** be static cost vectors for the same arena associated with adapters **A** and **B** respectively then the costs of a compound adapter are obtained by vector addition.

$$\frac{\mathbf{A} \triangleright \mathbf{x} \quad \mathbf{B} \triangleright \mathbf{y}}{\mathbf{A};\mathbf{B} \triangleright \mathbf{x} + \mathbf{y}} \qquad \frac{\mathbf{A} \triangleright \mathbf{x} \quad \mathbf{B} \triangleright \mathbf{y}}{\mathbf{A} \| \mathbf{B} \triangleright \mathbf{x} + \mathbf{y}}$$

2.6 Variable Costs

Variable costs are a function of channel through put. For variable cost purposes we model an adapter as two channels a forward and back channel. For an adapter of type A ↔ B: U the forward channel consists of all the read channels in signature A and all the writes is signature Band the back channel consists of all the reads in signature B and all the writes in signature A. The number of calls on the output side of the forward (respectively backward) channel is a function of the number of calls on the input side. The cost description of the forward (resp. backward) channel is a pair of functions (multiplier, cost) , the first of which expands the number of input calls to a number of output calls and the second of which translates the number of input into a cost. If we consider just the forward direction and sequential composition for simplicity, then:-

$$\frac{\mathbf{A} \triangleright [f_A, g_A] \quad \mathbf{B} \triangleright [f_B, g_B]}{\mathbf{A};\mathbf{B} \triangleright [f_B \circ f_A, g_A \hat{+} g_B \circ f_A]}$$

where $a \hat{+} b = l\ x.a(x) + b(x)$

Given the estimated number of calls in the forward direction for an adapter **A** as n_A and **A** ▷ [f, g] then the cost of **A** is $g(n_A)$.

The back channel constraints follow a similar structure. We denote a back channel constraint in the same way. The back channel constraint is denoted B_{back}.

$$\frac{\mathbf{A} \triangleright_{back} [f_A, g_A] \quad \mathbf{B} \triangleright_{back} [f_B, g_B]}{\mathbf{A};\mathbf{B} \triangleright_{back} [f_A \circ f_B, g_B \hat{+} g_A \circ f_B]}$$

Given the estimated number of calls in the backward direction for an adapter **A** as n_A and **A** B_{back} [f, g] then the cost of **A** is $g(n_A)$.

Note that if the number of calls in one channel, say the back channel, is a function of the number of calls in the other, in this case forward channel, the cost is accounted for totally by the cost function on the forward channel.

The total variable cost for an adapter is the sum of the forward channel and backwards channel costs.

Parallel composition add the multiplier and cost functions of the respective adapters:-

$$\frac{\mathbf{A} \triangleright [f_A, g_A] \quad \mathbf{B} \triangleright [f_B, g_B]}{\mathbf{A} \| \mathbf{B} \triangleright [f_B \hat{+} f_A, g_A \hat{+} g_B]} \qquad \frac{\mathbf{A} \triangleright_{back} [f_A, g_A] \quad \mathbf{B} \triangleright_{back} [f_B, g_B]}{\mathbf{A} \| \mathbf{B} \triangleright_{back} [f_B \hat{+} f_A, g_A \hat{+} g_B]}$$

2.7 Type Rules

2.7.1 Types

There are many possible models of types for adapter. In essence an adapter is a process that transforms actions on one interface into actions on other interfaces. The finest model of typing makes processes the same type if and only if they exhibit the same behaviour. The courses model of typing identifies processes as being of the same type if they communicate over the same channels. The model of typing adopted depends on the use that we wish to make of it. The model adopted below views adapters as processes that transform actions between interfaces which are collections of channels. This model is used to discuss composing adapters with one another to form adapter chains.

Adapter types are built up from the notion of interaction over type channels.

A **channel** is a directional communication port to a process. A channel has a direction, a name and an associated data type. A naming convention is assumed to exist to give channels an assumed semantics based on its name. Thus, for example, a channel with the name Video may have the specific semantics of being the video port of a process (whatever *that* may mean).

The syntax for a channel is “name direction type” where direction is one of “!” for *out* or “?” for *in*.

name !type	**An output channel**
name ? type	**An input channel**

A **signature** is a set of channels.

{ channel ,…,channel }	**A signature**

An **interface** is a name and a signature.

name: signature	**An interface**

Given a channel a !t the channel (respectively a ! t), the channel a ! t is called the complementary (respectively a ? t) channel. Given a channel **c** the operation **c*** gives the complementary channel.

This notion of complement is extended to signatures and interfaces. Given a signature **s**, **s*** the complement of **s**, is obtained by complementing each channel in **s** e.g.

{a, b, c}*	**{a*, b*, c*}**

Given an interface name **n** the complementary interface name is **n*.** Given an interface the complementary interface is obtained by complementing the name and complementing the signature e.g. given the interface **n:{**a ! t , b ? s } the interface the complementary interface is **n*:{**a ? t , b ! s }.

(name: signature)*	**name*: signature***

Communication takes place between complementary interfaces.

An **adapter** is a process that transforms one signature into another possibly requiring the use of some helper interfaces, e.g.:-

signature ↔ signature **: set of interfaces**	**A simple adapter type**

The "input" (left) side of the adapter is interpreted as a complementary signature.

In some cases we need to be able to specify a family of adapters that are parametric in their behaviour with respect to some of their channel types. Such a family is specified by allowing type variables to be used in place of actual types in channel type descriptions. These variables may be restricted to be subtypes of a particular type and must be substituted for uniformly throughout an adapter type. Such types are called polymorphic adapter types and are written as:-

$\prod_{a \prec T} A(a)$ where A is a, possibly polymorphic, adapter type with free variable α.

2.7.2 Sum and Product Data Types

The data type language is assumed to support sums and products of simple data types. In practice it is assumed that both the sums and products are represented by label rather than positional notation although for ease of discussion we will occasionally ignore this nicety.

When we wish to emphasise the "label" aspects of data type products and sums they will be written:-

- $\langle$ a :A, b :B, c :C $\rangle$ for a product with labels a, b and c and
- a :A | b :B | c :C for a sum with labels a, b and c
- an element of a product type will be written $\langle$ a = x, b = y, c = z $\rangle$ and
- an element of a sum will be written a(x) for the "a^{th}" element

Therefor channel types may be written e.g.

- α! $\langle$ a :A, b :B, c :C $\rangle$
- α! a :A | b :B | c :C
- α? $\langle$ a :A, b :B, c :C $\rangle$
- α? a :A | b :B | c :C

2.7.3 Subtypes

For output channels a channel ch!σ is a subtype of ch!τ if $\sigma \subseteq \tau$, that is a subtype of an output channel produces outputs compatible with the original type. For input channels a channel ch?σ is a subtype of ch?σ when $\tau \subseteq \sigma$, that is an input channel subtype must accept any input that the original channel type would accept.

A signature S is a subtype of a signature S' when S and S' have the same channel names with the same direction and the types of the output channels of S are subsets of the types of the output channels of S' and the types of the input channels of S are supersets of the types of the input channels of S' i.e. :-

$$Out(S) = Out(S') \wedge In(S) = In(S')$$

and

$$(\forall x \in Out(S).\, type(x) \subseteq type(x')) \wedge \forall x \in In(S).\, type(x) \supseteq type(x')$$

We write this relation as $S \leq S'$.

An interface n:s is a subtypes of an interface m:t when n=m and $s \leq t$.

We say one adapter type $A_1 \leftrightarrow A_2 : I_1, \ldots, I_n$ is a subtype of an adapter type $B_1 \leftrightarrow B_2 : I_1', \ldots, I_n', I_{n+1}, \ldots, I_k$ when $A_1 \leq B_1$ and $A_2 \leq B_2$ and $I_1 \leq I_1', \ldots, I_n \leq I_n'$. This is written as $A_1 \leftrightarrow A_2 : I_1, \ldots, I_n \leq B_1 \leftrightarrow B_2 : I_1', \ldots, I_n', I_{n+1}, \ldots, I_k$.

(**Aside**: At some stage we will need to extend these notions to adapters that convert between n "input" processes and m "output" processes e.g. in modelling replication. The notation used will be $A_1 \times \ldots \times A_n \leftrightarrow B_1 \times \ldots \times B_m : I_1, \ldots, I_k$.)

2.7.4 Type Conformance

Type conformance arises for channels when matching an output channel against an input channel.

Generally the matching constraint between complementary channels, e.g. ch!τ and ch?σ, is that $\tau \subseteq \sigma$. That is, the values output from the writing process are contained in the values readable to the reading process.

Signature matching therefore requires this behaviour for all channels of the signature That is, S and S' are conformant if and only if S and S' provide the same channel names i.e.

$$Out(S) = Out(S') \wedge In(S) = In(S')$$

and the channels are type conformant:-

$$(\forall x \in Out(S) \cap In(S').type(x) \subseteq type(x')) \wedge \forall x \in In(S) \cap Out(S').type(x') \subseteq type(x)$$

where Out(X) is the set of output channels names of signature X and In(X) is the set of input channels names of signature X.

Conformance of signatures S and S' is written $S \approx S'$.

2.7.5 Type Rules

Type rules describe how adapter channels compose. The simplest case is allowing exactly matching adapters to be composed:-

$$\frac{\mathbf{A} \triangleright A \leftrightarrow B{:}U \quad \mathbf{B} \triangleright B \leftrightarrow C{:}V}{\mathbf{A};\mathbf{B} \triangleright A \leftrightarrow C{:}U \cup V}$$

$$\frac{\mathbf{A} \triangleright A \leftrightarrow A'{:}U \quad \mathbf{B} \triangleright B \leftrightarrow B'{:}V}{\mathbf{A}||\mathbf{B} \triangleright A \cup B \leftrightarrow A' \cup B'{:}U \cup V}$$

More generally, for sequential composition, it is possible to compose two adapters that are conformant.

$$\frac{\mathbf{A} \triangleright A \leftrightarrow B{:}U \quad \mathbf{B} \triangleright B' \leftrightarrow C{:}V}{\mathbf{A};\mathbf{B} \triangleright A \leftrightarrow C{:}U \cup V}$$ where B ≈ B'

2.7.6 Subtype Polymorphism

Universal polymorphism allows the expression of fully generic adapter types. We introduce polymorphic adapter types to express the idea of an adapter that behaves in the same way irrespective of the details of some of the type parameter used in the type expression provided that this type parameter is uniformly replaced throughout the expression. Parametric types may be restricted to range over subtypes of some particular type. We notate a polymorphic adapter type as:-

$\prod_{a \prec T} A(a)$ where A is a, possibly polymorphic, adapter type with free variable α.

e.g.

$\prod_{a \prec T} A(a) \leftrightarrow B(a){:}U(a)$ where α ranges over subtypes of T.

An adapter of this type is said to be polymorphic in α. This means that any subtype of T may be uniformly substituted for α to create an instance. That is, it obeys the elimination rule:-

$$\frac{\mathbf{A} \triangleright \prod_{a \prec T} A(a) \leftrightarrow B(a){:}U(a)}{\mathbf{A}_t \triangleright A(t) \leftrightarrow B(t){:}U(t)}$$ where t is a sub type of T.

i.e. it obeys the general rule:-

$$\frac{\mathbf{A} \triangleright \prod_{a \prec T} A(a)}{\mathbf{A}_t \triangleright A(t)}$$ where A is a possibly polymorphic adapter type and t is a sub type of T.

As an example of a polymorphic adapter consider the encryption adapter type:-

$$\prod_{a \prec ByteStream} data!a \leftrightarrow data!Encrypt(a){:}Key?keyType$$

This is the type of an adapter that will encrypt any ByteStream subtype assuming that a helper interface is available to provide a key of type keyType.

Polymorphic adapters may be composed. The restriction the subtype restrictions on the common type variables are inherited from both subtype constraints. The derived rule for one variable is:-

$$\frac{\mathbf{A} \triangleright \prod_{a \prec T} A(a) \leftrightarrow B(a):U(a) \quad \mathbf{B} \triangleright \prod_{b \prec T'} B(b) \leftrightarrow C(b):V(b)}{\mathbf{A}\,;\mathbf{B} \triangleright \prod_{g \prec T \cap T'} A(g) \leftrightarrow C(g):U(g) \cup V(g)}$$

where T ∩ T' is required to be non-empty and γ is a new variable not occurring free in A, C, U or V.

2.7.7 Sum and Product (Record) Polymorphism

A further operator is introduced for each of the sum and record. If τ and σ are sum types then τ ⊕ σ is a sum type and if t and s are product types then τ ⊗ σ is a product type. These operations are referred to as tensor sum and tensor product respectively.

Two sum (product) types are said to be compatible if they have distinct labels.

Tensor sum joins two compatible sum types together by merging them into a single sum type. Tensor product joins two compatible product types together by merging them into a single product type.

Given a channel type α! ⟨ a :A, b :B ⟩ and a type variable τ then α! ⟨ a :A, b :B ⟩ ⊗ τ expresses a channel that matches channel whose type is a record containing a of type A and b of type B.

To complete the picture the types **0** stand for the empty sum type and **1** for the empty product type and the laws:-

$$a \oplus b = b \oplus a \qquad a \otimes b = b \otimes a$$
$$(a \oplus b) \oplus c = a \oplus (b \oplus c) \qquad (a \otimes b) \otimes c = a \otimes (b \otimes c)$$
$$a \oplus \mathbf{0} = a \qquad a \otimes \mathbf{1} = a$$

The presence of **0** and **1** means that a channel such as α! ⟨ a :A, b :B ⟩ ⊗ τ can be used to match α! ⟨ a :A, b :B ⟩ or any extension.

2.7.8 Widening Adapters

The parallel composition of adapters offers the opportunity for another kind of subtyping relation. An adapter type A'may be considered a subtype of an adapter A when A'= A || I where I is an identity map on additional channels in both the forward and backward directions so that the type of I

is $X \leftrightarrow X$ for some signature X. That is, any channels additional to those in A are passed through unaltered. We refer to this operation as "padding". Padding is used to give adapters the "right shape" for sequential composition.

2.7.9 Interaction Channels

An interaction channel is a packaging of channels into an RPC model of interaction. An RPC call is modelled as an "out" channel with a product data type representing the arguments to the RPC sequenced with an "in" channel with a sum type representing the possible returned results (including exceptions). The interaction type extends the adapter typing by a pair of functions that map input side incoming interaction channel calls to output side outgoing interaction channels and output side incoming interaction channel calls to input side outgoing interaction channels calls.

The type of an interaction channel is a channel name, an argument type which is a product and a return type which is a Sum. Interaction types are directed in that they are either called and then return (called an incoming interaction channel) or they call and have results returned to them (called outgoing interaction channels).

name ? Product type ! Sum	**Incoming Interaction Channel**
name ! Product type ? Sum	**Outgoing Interaction Channel**

The obvious notion of complementary interaction channel types is adopted and type conformance and subtyping are defined in the obvious way. That is interaction channels types $n!\tau?\sigma$ and $n?\tau!\sigma'$ are conformant when $\tau \subseteq \tau'$ and $\sigma' \subseteq \sigma$. An outgoing channel $n!\tau?\sigma'$ is a subtype of another outgoing channel $n!\tau?\sigma$ if $\tau' \subseteq \tau$ and $\sigma \subseteq \sigma'$ and an incoming channel $n?\tau!\sigma'$ is a subtype of an incoming channel $n?\tau!\sigma$ when $\sigma' \subseteq \sigma$ and $\tau \subseteq \tau'$.

Interaction channels can then be simply incorporated into the above frame work of signatures, interfaces etc.

3 Building An Adapter chain

3.1 Goals

An adapter chain is built from basic adapters to meet a goal. The goal is described in terms of the type of transformation the adapter is required to carry out, the acceptable cost of the adapter (as a sum of the static and variable costs) and any static constraints that the adapter must meet. This goal is stated with respect to a set of assumptions that define the expected number of calls made to the adapter in both the forward and backward direction and any adapter constraints that may be assumed to be already met by the process.

This section modifies the framework above in "A Formalism for Perspectives" to take account of the additional structure of a goal.

Assumptions are represented as a pair of integers representing the expected number of forward calls and expected number of backward calls, and a set of atomic propositions representing the any adapter constraints already satisfied by the process.

A goal is a type, a cost constraint and a set of static constraints to be met by the adapter.

An adapter type may be satisfied by any of its subtypes.

For simplicity cost constraints will be assumed to be of the form cost $\leq$ $cost_{max}$ so that cost goals are simply described as cost vectors representing $cost_{max}$.

Static constraints are propositional formulas.

The goal under assumptions is written (n, m, a) $\Rightarrow$ (τ,**v**, s) where n is the number of calls expected in the forward direction, m is the number of calls expected in the backward direction, a is a set of atomic propositions, t is a type, v is a cost vector and s is a set of propositional formulas. The problem is to find **A** such that **A** satisfies (n, m, a) $\Rightarrow$ (τ, **v**, s) i.e. **A** satisfies (τ, **v**, s) under assumptions (n, m, a).

Define **A** $\triangleright$ (n, m, a) $\Rightarrow$ (τ, **v**, s) by

- **A** $\triangleright_{\text{type}}$ τ'where is a τ'subtype of τ.
- **c** + (f n) + (b m) ≤ **v** where **A** $\triangleright_{\text{cost}}$ **c, A** $\triangleright_{\text{forward}}$ [f, g] and **A** $\triangleright_{\text{backward}}$ [b, h]
- **A** $\triangleright_{\text{static}}$ Ù s where Ù s is the conjunction of the formulas of s
- **A** $\triangleright_{\text{adapter}}$ (a, w) where w is any consistent set of atomic adapter propositions

$$\frac{\begin{array}{c} \mathbf{A} \triangleright_{\text{cost}} \mathbf{c} \\ \mathbf{A} \triangleright_{forward} [f, g] \\ t' \leq t \qquad \mathbf{A} \triangleright_{backward} [b, h] \\ \mathbf{A} \triangleright_{type} t' \quad \mathbf{c} + f(n) + b(m) \leq \mathbf{v} \quad \mathbf{A} \triangleright_{static} \wedge s \quad \mathbf{A} \triangleright_{adapter} (a, w) \end{array}}{\mathbf{A} \triangleright (n, m, a) \Rightarrow (t, \mathbf{v}, s)}$$

where w is a consistent set of atomic propositions.

4 Arenas

4.1 Introduction

An arena is the context that is used to describe an adapter. It defines the sets of propositional letters used to define the static context and the adapter context, the set of cost vectors used for fixed costs, the set of cost and multiplier functions used for the variable costs and the data types available for the adapter types.

In practice many aspects of an arena will be defined implicitly. For example where as the set of propositional letters used for context constraints might be explicitly defined the set of data types available may be defined as any type available in a given programming language. Below we define an arenas to be used for Java implementations of adapters.

4.2 Java Based Arenas

The static constraints are defined as a set of propositional letters. The adapter (history) constraints are defined as a set of propositional letters.

The static cost vector is specified as a class that supports addition. The variable cost description is given as a class. This class provides forward and backwards cost and multiplier functions. The cost functions are of type integer (i.e. the number of calls on the forward or backward channels) to the static cost type, a multiplier functions are of type integer to integer. The class also supports an addition function for combining objects of this class.

The data types are taken to any Java data type.

Arena ::=
Arena name *[* **subtypes** Arena *]* {
*[***static constraints** propositional_letter**]*
*[***adapter constraints** propositional_letter **]*
*[***static costs** class_name*]*
*[***variable costs** class_name*]* }

5 Adapters

5.1 Java Based Adapters

Notes:

Java based adapters have a typing based entirely on interaction channels.

For an adapter constraints pair (α,β) we call α the requirements and β the assertions.

Interfaces are generic in that they are parameterised by type variables that may be restricted to subtypes of a particular type in the formal parameter list. The syntax for a type variable is "type name" where if type is present it restricts the variable "name" to range over subtypes of "type".

In adapters type variables are used to bind together behaviour across interfaces. The formal parameters of an adapter define the type variables used in the adapter across its interfaces.

Although an interaction channel declaration looks like an ordinary method declaration it is actually implemented as method of one argument. The single argument is a record containing fields with the same names as the arguments. A form of "record polymorphism" is adopted to allow adapter methods to be called with more arguments than they are declared with. The additional arguments are copied over from input interaction channels to output interaction channels. This polymorphism mirrors the use of the parallel adapter composition to compose an adapter with parallel identity to pad an adapter out to the "right shape" for a sequential composition.

The &rest parameter option in the argument list of an interaction channel allows the ability to specify the channel will match any extension of the channel (similarly with the &rest parameter in the exceptions list). The semantics of adapters is expected to treat these additional parameters "uniformly" i.e. in a way not dependent on their actual type.

Interface ::=
Interface name *[*(formal_parameter_list)*] [***subtypes** Interface*]* {
*(***incoming interaction channel** interaction_channel *)**
*(***outgoing interaction channel** interaction_channel*)** }

formal_parameter_list = (type name *(*, type name*)*)

type ::= name | type name

interface ::= name *[*actual_parameter_list*]*

actual_parameter_list ::= type *(*, type*)**

interaction_channel ::= *[***abstract***]* type name (arg_list *[*&rest *[*type*]* name*]*) *(***exception** exception*)** *[*&rest *[*type*]* name*]*

exception ::= type name

Adapter ::=
Adapter name *[*(formal_parameter_list)*]*
*[***built for** Arena | **subtypes** Adapter*]*
*(***from interface** interface **to interface** interface
*[***helpers** interface_list*])**{
*(***static cost** cost object*)**
*(***variable cost** cost object*)**
*(***static constraint** propositional expression*)**
*(***left adapter constraint** propositional_letter**)**
*(***adapter assertions** atomic_proposition**)**
}