MODELLING DISTRIBUTED SYSTEMS USING Z

Howard Bowman and John Derrick
Computing Laboratory, University of Kent, Canterbury, UK.

Abstract

The ODP development model is a natural progression from OSI. Multiple viewpoints are used to specify complex ODP systems. Formal methods are playing an increasing role within ODP. There are two technical problems concerning the use of formal techniques within ODP which have yet to be addressed: these are unification and consistency checking. We show how Z can be used to provide a solution for both; and hence provide a mechanism for Z to be used properly in the ODP development process.

Keywords: ODP, Z, Consistency, Viewpoints.

Introduction

The use of formal methods outside academic institutions has not penetrated industry in the manner that many have been predicting. However, there are two areas that formal methods have been making a significant impact, these are in *standards* and *safety-critical systems*. This paper discusses the implications and integration of formal techniques, in particular Z, into the Open Distributed Processing (ODP) standard initiative.

The ODP standardization initiative is a natural progression from OSI, broadening the target of standardization from the point of interconnection to the end-to-end system behaviour. The objective of ODP [4] is to enable the construction of distributed systems in a multi-vendor environment through the provision of a general architectural framework that such systems must conform to. One of the cornerstones of this framework is a model of multiple viewpoints which enables different participants to observe a system from a suitable perspective and at a suitable level of abstraction [6]. There are five separate viewpoints presented by the ODP model: Enterprise, Information, Computational, Engineering and Technology. Requirements and specifications of an ODP system can be made from any of these viewpoints.

Formal methods are playing an increasing role within ODP (Part 4 of the ODP-RM outlines requirements for applying formal description techniques in the specification of ODP systems.), and we aim to provide a mechanism by which specific techniques can be used within ODP. The suitability of a wide spectrum of FDTs is currently being assessed (eg LOTOS, Estelle, SDL, Z, Object-Z and RAISE). Amongst these LOTOS and Z are becoming dominant. The first compliant ODP specification, the Trader, is being written using Z for the information and computational viewpoint.

However, while it has been accepted that the viewpoint model greatly simplifies the development of system specifications and offers a powerful mechanism for handling diversity within ODP, the practicalities of how to make the approach work are only beginning to be explored. In particular, one of the consequences of adopting a multiple viewpoint approach to development is that descriptions of the same or related entities can appear in different viewpoints and must co-exist. *Consistency* of specifications across viewpoints thus becomes a central issue. However, the actual mechanism by which consistency can be checked and maintained is only just being addressed [9, 3]. In particular, although Z is being used as a viewpoint specification language in ODP, there is as yet no mechanism to describe the combination of different Z viewpoint specifications, or the consistency of them.

In Section 2 we develop the unification mechanism for Z specifications. In Section 3 we present an example of the technique by specifying the dining philosophers problem using viewpoints. Section 4 discusses consistency checking of viewpoint specifications, and we make some concluding remarks in Section 5.

Unification in Z

Given a model of multiple viewpoints, descriptions of the same or related entities can appear in different viewpoints and must co-exist. Clearly the different viewpoints must be consistent, i.e. the properties of one viewpoint specification do not contradict those of another. In addition, during the development process there must be some way to combine specification from different viewpoints into a single implementation specification. This process of combining two specifications is known as *unification*. Furthermore, the unification of two specifications must be a refinement of both, [1]. Unification can also be used, because of this common refinement, as a method by which to check consistency. To check the consistency of two specifications, we check for contradictions within the unified specification.

Given a refinement relation, $\sqsubseteq$, defined in a formal specification technique, we can define the unification $\mathcal{U}(T_1, T_2)$ of two specifications as $\{T : T_1, T_2 \sqsubseteq T$ and if $T_1, T_2 \sqsubseteq S$ then $T \sqsubseteq S\}$. Unification of Z specifications will therefore depend upon the Z refinement relation, which is given in terms of two separate components - data refinement and operation refinement, [7].

Z is a state based FDT, and Z specifications consist of informal English text interspersed with formal mathematical text. The formal part describes the abstract state of the system (including a description of the initial state of the system), together with the collection of available operations, which manipulate the state. We assume the reader is familiar with details of the language and its refinement relation, introductionary texts include [7, 8].

The unification algorithm we describe is divided into three stages: normalization, common refinement (which we usually term unification itself), and re-structuring. Normalization identifies commonality between two specifications, and re-writes the specifications into normal forms suitable for unification. Unification itself takes two normal forms and produces the least refinement of both. Because normalization will hide some of the specification structure introduced via the schema calculus, it is necessary to perform some re-structuring after unification to re-introduce some of the specifiers style. We do not discuss re-structuring here.

Normalization

Given two different viewpoint specifications of the same (ODP) system, the commonality between the specifications needs to be identified. These will be given by co-viewpoint mappings that describe the naming, and other, conventions in force. Once the commonality has been identified, the appropriate elements of the specifications are re-named.

Normalization will also expand data-type and schema definitions into a normal form. The purpose of normalization is to hide the structuring of schemas (which is needed in order to provide automatic unification techniques) and expand declarations into maximal type plus predicate declarations. For example, normalization of a declaration part of a schema involves replacing every set X which occurs in a declaration $x : X$, with its corresponding maximal type and adding predicates to the predicate part of the schema involved to constrain the variable appropriately.

Normalization also expands schemas defined via the schema calculus into their full form. All schema expressions involving operations from the schema calculus can be expanded to a single equivalent vertical schema. Examples of this type of normalization are given in [7].

State Unification

The purpose of state unification is to find a common state to represent both viewpoints. The state of the unification must be a data refinement of the state of both viewpoints. Furthermore, it should be the least refinement whenever possible. This is needed to ensure we do not add too much detail during unification because additional detail might add inconsistencies that were not due to inconsistencies in the original viewpoint specifications.

The essence of all constructions will be as follows. If an element x is declared in both viewpoints as $x : T_1$ and $x : T_2$ respectively, then the unification will include a declaration $x : T$ where T is the least refinement of T_1 and T_2. The type T will be the smallest type which contains a copy of both T_1 and T_2. For example, if T_1 and T_2 can be embedded in some maximal type then T is just the union of $T_1 \cup T_2$. We will prove the correctness of this unification below.

Given two viewpoint specifications containing the following fragment of state description:

$$\begin{array}{|l}\multicolumn{1}{l}{D} \\ \hline x : S \\ \hline pred_S \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{D} \\ \hline x : T \\ \hline pred_T \\ \hline\end{array}$$

we unify as follows

$$\begin{array}{|l}\multicolumn{1}{l}{D} \\ \hline x : S \cup T \\ \hline x \in S \implies pred_S \\ x \in T \implies pred_T \\ \hline\end{array}$$

whenever $S \cup T$ is well founded. (Axiomatic descriptions are unified in exactly the same manner.) This representation is needed in order to preserve the widest range of possible behaviours.

Operation Unification

Once the data descriptions have been unified, the operations from each viewpoint need to be defined in the unified specification. We assume all renaming of names visible to the environment has taken place. Unification of schemas then depends upon whether there are duplicate definitions. For operations defined in just one of the viewpoint specifications, these are included in the unification with appropriate adjustments to take account of the unified state.

For operations which are defined in both viewpoint specifications, the unified specification should contain an operation which is the least refinement of both, with respect to the unified representation of state. The unification of two operations is defined via their pre- and post-conditions. Given a schema it is always possible to derive their pre- and post-conditions, [5]. Given two schemas A and B representing operations, both applicable on some unified state, then the unification of A and B is:

$$\begin{array}{|l}\multicolumn{1}{l}{\mathcal{U}(A, B)} \\ \hline \vdots \\ \hline pre\ A \vee pre\ B \\ pre\ A \implies post\ A \\ pre\ B \implies post\ B \\ \hline\end{array}$$

where the declarations are unified in the manner of the preceding subsection. This definition ensures that if both preconditions are true, then the unification will satisfy both post-conditions. Whereas if just one pre-condition is true, only the relevant post-condition has to be satisfied.

Unification is the least refinement

To show that unification is correct, we must show that it is the least refinement of the two viewpoint specifications. We sketch a proof here, showing first it is a refinement, before showing that any other refinement will also refine the unification. As above we decorate elements in viewpoint j with a subscript j. Given two fragments of viewpoint specifications below, both with the state described by the schema D, and an operation A which manipulates the state D and possibly other non-overlapping portions of the state.

$$\begin{array}{|l} D \\ \hline x : S \\ \hline pred_S \\ \hline \end{array} \qquad \begin{array}{|l} D \\ \hline x : T \\ \hline pred_T \\ \hline \end{array}$$

$$\begin{array}{|l} A \\ \hline \vdots \\ OpD \\ \hline pre\ A \\ post\ A \\ \hline \end{array} \qquad \begin{array}{|l} A \\ \hline \vdots \\ OpD \\ \hline pre\ A \\ post\ A \\ \hline \end{array}$$

where Op is either Δ, Ξ or blank. The unification is

$$\begin{array}{|l} D \\ \hline x : S \cup T \\ \hline x \in S \implies pred_S \\ x \in T \implies pred_T \\ \hline \end{array} \qquad \begin{array}{|l} A \\ \hline \vdots \\ OpD \\ \hline pre\ A_1 \lor pre\ A_2 \\ pre\ A_1 \implies post\ A_1 \\ pre\ A_2 \implies post\ A_2 \\ \hline \end{array}$$

To describe the refinement, the retrieve relation R_1 between the unification and viewpoint one is given by

$$\begin{array}{|l} R_1 \\ \hline D \\ D_1 \\ \hline x_1 = \{x\} \cap S \\ \hline \end{array}$$

Then it is easy to see that: $pre\ A = pre\ A_1 \lor pre\ A_2$, $pre\ A_1 \land R_1 \implies pre\ A$, $pre\ A_1 \land \Delta R_1 \land A \Rightarrow post\ A_1$. Hence, the unification is indeed a common refinement. We now show that it is the least refinement.

To do so suppose that we are given another refinement of both viewpoints, which is described by state E and operation B. Then there exist retrieve relations ED_1 and ED_2 such that:

$$pre\ A_1 \land ED_1 \Rightarrow pre\ B, \quad pre\ A_1 \land \Delta ED_1 \land B \Rightarrow post\ A_1$$
$$pre\ A_2 \land ED_2 \Rightarrow pre\ B, \quad pre\ A_2 \land \Delta ED_2 \land B \Rightarrow post\ A_2$$

We then require a retrieve relation X between D and E such that $pre\ A \land X \implies pre\ B$, $pre\ A \land \Delta X \land B \implies post\ A$. Letting $X = ED_1 \land ED_2$ will suffice. Since $pre\ A = pre\ A_1 \lor pre\ A_2$, we have the following deduction: $pre\ A \land (ED_1 \land ED_2)$ equals $(pre\ A_1 \lor pre\ A_2) \land (ED_1 \land ED_2)$ equals $(pre\ A_1 \land ED_1 \land ED_2) \lor (pre\ A_2 \land ED_2 \land ED_1)$ implies $(pre\ B \land ED_2) \lor (pre\ B \land ED_1)$ implies $pre\ B \land (ED_1 \lor ED_2)$ implies $pre\ B$.

For the second deduction we have: $pre\ A \land \Delta(ED_1 \land ED_2) \land B$ equals $(pre\ A_1 \lor pre\ A_2) \land \Delta ED_1 \land \Delta ED_2) \land B$ implies $(post\ B \land \Delta ED_2) \lor (post\ B \land \Delta ED_1)$ equals $post\ B \land (\Delta ED_1 \lor \Delta ED_2)$, and hence $post\ B$ follows. Therefore, the state E and operation B are refinements of the unification.

Example

To illustrate unification with Z, we shall consider the following viewpoint specifications of the dining philosophers problem. The dining philosophers problem, [2], is a classic problem in synchronization. A group of N philosophers sit round a table, laid with N forks. There is one fork between each adjacent pair of philosophers. Each philosopher alternates between thinking and eating. To eat, a philosopher must pick up its right-hand fork and then the left-hand fork. A philosopher cannot pick up a fork if its neighbour already holds it. To resume thinking, the philosopher returns both forks to the table. We shall describe the problem via two viewpoint Z specifications, each representing a particular concern. We shall then describe their unification.

The Philosophers Viewpoint

This viewpoint considers the specification from the point of view of the philosophers. There are N philosophers who are either thinking, eating or holding their right fork. Note that since the latter is just a state of mind (for a philosopher!) there is no need to describe the operations from a forks point of view at all in this viewpoint. In order to correctly describe the synchronization, the philosophers have to be aware of the fork's existence, and some of the possible states a fork can be in.

$$\begin{array}{|l} N : \mathbb{N} \\ \hline N > 1 \end{array}$$

$$tabled == 1..N$$
$$PhilStatus ::= Thinking \mid HasRightFork \mid Eating$$
$$[ForkStatus]$$
$$\mid Free \in ForkStatus$$

Then the system from the philosophers point of view is just defined by the state of the philosopher, however, there is an awareness of the forks existence.

$$\begin{array}{|l} Table \\ \hline phils : tabled \rightarrow PhilStatus \\ forks : 1..N \rightarrow ForkStatus \\ \hline \end{array}$$

And initially the philosophers are all thinking, and we make no constraints on forks.

$$\begin{array}{|l} InitTable \\ \hline Table \\ \hline \operatorname{ran} phils = \{Thinking\} \\ \hline \end{array}$$

We can now describe the operations available. A thinking philosopher can pick up its right-hand fork, it also knows that to do this the right hand fork must be free.

$$\begin{array}{|l} GetRightFork \\ \hline \Delta Table \\ n? : tabled \\ \hline forks(n? \bmod N + 1) = Free \\ phils(n?) = Thinking \\ phils' = phils \oplus \{n? \mapsto HasRightFork\} \\ \hline \end{array}$$

Philosophers who hold their right fork can begin eating upon picking up their left-hand fork, whenever it is free.

$$
\begin{array}{|l}
\textit{GetLeftFork} \\
\hline
\Delta \textit{Table} \\
n? : \textit{tabled} \\
\hline
\textit{forks}(n?) = \textit{Free} \\
\textit{phils}(n?) = \textit{HasRightFork} \\
\textit{phils}' = \textit{phils} \oplus \{n? \mapsto \textit{Eating}\} \\
\hline
\end{array}
$$

Finally to resume thinking, a philosopher releases both forks.

$$
\begin{array}{|l}
\textit{DropForks} \\
\hline
\Delta \textit{Table} \\
n? : \textit{tabled} \\
\hline
\textit{phils}(n?) = \textit{Eating} \\
\textit{phils}' = \textit{phils} \oplus \{n? \mapsto \textit{Thinking}\} \\
\hline
\end{array}
$$

The Forks Viewpoint

This viewpoint considers the specification from the point of view of the forks. There are N forks each of which is either free or busy. The fact that the philosopher might change state when a fork is picked up or dropped does not concern forks.

$$
\begin{array}{|l}
N : \mathbb{N} \\
\hline
N > 1
\end{array}
$$

$$
\begin{array}{l}
\textit{ForkStatus} ::= \textit{Free} \mid \textit{Busy} \\
[\textit{PhilStatus}] \\
\mid \textit{Thinking}, \textit{HasRightFork}, \textit{Eating} \in \textit{PhilStatus}
\end{array}
$$

Then the system from the forks point of view is just defined by the state of the fork.

$$
\begin{array}{|l}
\textit{Table} \\
\hline
\textit{forks} : 1..N \rightarrow \textit{ForkStatus} \\
\textit{phils} : \textit{tabled} \rightarrow \textit{PhilStatus} \\
\hline
\end{array}
$$

And initially the forks are all free.

$$
\begin{array}{|l}
\textit{InitTable} \\
\hline
\textit{Table} \\
\hline
\operatorname{ran} \textit{forks} = \{\textit{Free}\} \\
\hline
\end{array}
$$

We can now describe the operations available. A free fork can be picked up. Note that clearly the specifier of this viewpoint has to be aware that the forks have a polarity for the object that picks them up, and thus describes the operations in terms of that polarity (alternatively, this polarity could be described via co-viewpoint mappings).

$$
\begin{array}{|l}
\textit{GetRightFork} \\
\hline
\Delta \textit{Table} \\
n? : 1..N \\
\hline
\textit{phils}(n?) = \textit{Thinking} \\
\textit{forks}(n? \bmod N + 1) = \textit{Free} \\
\textit{forks}' = \textit{forks} \oplus \{n? \bmod N + 1 \mapsto \textit{Busy}\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{GetLeftFork} \\
\hline
\Delta \textit{Table} \\
n? : 1..N \\
\hline
\textit{phils}(n?) = \textit{HasRightFork} \\
\textit{forks}(n?) = \textit{Free} \\
\textit{forks}' = \textit{forks} \oplus \{n? \mapsto \textit{Busy}\} \\
\hline
\end{array}
$$

Finally, both forks can be released.

$$
\begin{array}{|l}
\textit{DropForks} \\
\hline
\Delta \textit{Table} \\
n? : 1..N \\
\hline
\textit{phils}(n?) = \textit{Eating} \\
\textit{forks}' = \textit{forks} \oplus \{n? \mapsto \textit{Free}, n? \bmod N + 1 \mapsto \textit{Free}\} \\
\hline
\end{array}
$$

Unifying the Viewpoints

We can now describe the unification of these two viewpoints in terms of the algorithm given above. First all normalizations are undertaken. These will describe the declarations in normal form, and substitute expressions for all abbreviations. In the philosophers specification *tabled* is expanded, then the types normalized. So for example, the schema *GetRightFork* in the philosophers viewpoint is re-written as

$$
\begin{array}{|l}
\textit{GetRightFork} \\
\hline
\Delta \textit{Table} \\
n? : \mathbb{N} \\
\hline
1 \leq n? \leq N \\
\textit{forks}(n? \bmod N + 1) = \textit{Free} \\
\textit{phils}(n?) = \textit{Thinking} \\
\textit{phils}' = \textit{phils} \oplus \{n? \mapsto \textit{HasRightFork}\} \\
\hline
\end{array}
$$

The declarations in the unification are then:

$$
\begin{array}{|l}
N : \mathbb{N} \\
\hline
N > 1
\end{array}
$$

$$\textit{PhilStatus} ::= \textit{Thinking} \mid \textit{HasRightFork} \mid \textit{Eating}$$

$$\textit{ForkStatus} ::= \textit{Free} \mid \textit{Busy}$$

These declarations have been unified as described above. Next one schema called *Table* with initial state *InitTable* are built out of the component viewpoints. The combined declarations and predicates become:

$$
\begin{array}{|l}
\textit{Table} \\
\hline
\textit{phils} : \mathbb{P}(\mathbb{N} \times \textit{PhilStatus}) \\
\textit{forks} : \mathbb{P}(\mathbb{N} \times \textit{ForkStatus}) \\
\hline
\forall x : \mathbb{N} \bullet \exists_1 y : \textit{PhilStatus} \bullet (x, y) \in \textit{phils} \\
\forall x : \mathbb{N} \bullet \exists_1 y : \textit{ForkStatus} \bullet (x, y) \in \textit{forks} \\
\operatorname{dom} \textit{phils} = \operatorname{dom} \textit{forks} = 1..N \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{InitTable} \\
\hline
\textit{Table} \\
\hline
\operatorname{ran} \textit{phils} = \{\textit{Thinking}\} \\
\operatorname{ran} \textit{forks} = \{\textit{Free}\} \\
\hline
\end{array}
$$

The operations available which are represented by the schemas have to be unified. To do so we calculate their pre- and post-conditions, and produce a unification with respect to the unified state as represented by the schema *Table*. Upon simplifying, the operation schemas *GetRightFork* and *GetLeftFork* become:

GetRightFork

$\Delta Table$
$n? : \mathbb{N}$

$1 \leq n? \leq N$
$phils(n?) = Thinking$
$phils' = phils \oplus \{n? \mapsto HasRightFork\}$
$forks(n? \bmod N + 1) = Free$
$forks' = forks \oplus \{n? \bmod N + 1 \mapsto Busy\}$

GetLeftFork

$\Delta Table$
$n? : \mathbb{N}$

$1 \leq n? \leq N$
$phils(n?) = HasRightFork$
$phils' = phils \oplus \{n? \mapsto Eating\}$
$forks(n?) = Free$
$forks' = forks \oplus \{n? \mapsto Busy\}$

DropForks is unified in a similar fashion.

Checking Consistency

Consistency checking involves checking the unified specification for contradictions. Consistency checking consists of checking both the consistency of the state model and the consistency of all the operations. The nature of unification as the least refinement means that this involves checking the intersection of the two viewpoints in the unified state model, and the conjunction of the pre-conditions in each operation.

For example, consider the general form of state unification given in Section 2.2:

D

$x : S \cup T$

$x \in S \Longrightarrow pred_S$
$x \in T \Longrightarrow pred_T$

This state model is consistent as long as both $pred_S$ and $pred_T$ can be satisfied for $x \in S \cap T$. In the classroom example, suppose the class consisted of just the element 2, i.e. $d = \{2\}$. Both pre-conditions in the unified state, $d \in \mathbb{P}\{1,2\}$ and $d \in \mathbb{P}\{2,3,4\}$, now hold giving the state invariant $Min \leq \#d \leq Max$. Thus the consistency of the viewpoint specifications of the classroom requires that $Min \leq Max$. This type of consistency condition is called a *correspondence rule* in ODP, [4], that is a condition which is necessary but not necessarily sufficient to guarantee consistency.

Consistency checking also needs to be carried out on each operation in the unified specification. The definition of operation unification means that we have to check for consistency when both pre-conditions apply. That is, if the unification of A and B is denoted $\mathcal{U}(A,B)$, we have:

$$pre\ \mathcal{U}(A,B) = pre\ A \vee pre\ B,$$
$$post\ \mathcal{U}(A,B) = (pre\ A \Rightarrow post\ A) \wedge (pre\ B \Rightarrow post\ B)$$

So the unification is consistent as long as $(pre\ A \wedge pre\ B) \Rightarrow (post\ A = post\ B)$. In the classroom example, this amounts to checking the operation *Leave* when

$$(p? \in d \cap \{1,2\}) \wedge (p? \in d \cap \{2,3,4\} \wedge \#d > Min + 1)$$

In these circumstances, the two post-conditions are $d' = d \setminus \{p?\}$ and $d' = d \setminus \{p?, 2\}$. Now the two pre-conditions apply iff both $p? = 2$ and $Min < 0$, in which case the post-conditions are $d' = d \setminus \{2\}$ and $d' = d \setminus \{2\}$, and thus consistent. Hence, *Leave* is consistent (and therefore so are two viewpoint specifications) whenever the correspondence rule $Min \leq Max$ holds.

Conclusions

The use of viewpoints to enable separation of concerns to be undertaken at the specification stage is a cornerstone of the ODP model. However, the practicalities of how to make the approach work are only beginning to be explored. Two issues of importance are unification and consistency checking. Our work attempts to provide a methodology to undertake unification and consistency checking for Z specifications.

References

[1] G. Cowen, J. Derrick, M. Gill, G. Girling (editor), A. Herbert, P. F. Linington, D. Rayner, F. Schulz, and R. Soley. *Prost Report of the Study on Testing for Open Distributed Processing.* APM Ltd, 1993.

[2] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages.* Academic Press, 1968.

[3] K. Farooqui and L. Logrippo. Viewpoint transformations. In J. de Meer, B. Mahr, and O. Spaniol, editors, *2nd International IFIP TC6 Conference on Open Distributed Processing*, pages 352–362, Berlin, Germany, September 1993.

[4] ISO/IEC JTC1/SC21/WG7. *Basic reference model of Open Distributed Processing - Parts 1-4*, July 1993.

[5] S. King. Z and the refinement calculus. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90 VDM and Z - Formal Methods in Software Development*, LNCS 428, pages 164–188, Kiel, FRG, April 1990. Springer-Verlag.

[6] P. F. Linington. Introduction to the Open Distributed Processing Basic Reference Model. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 3–13, Berlin, Germany, September 1991. North-Holland.

[7] B. Potter, J. Sinclair, and D. Till. *An introduction to formal specification and Z.* Prentice Hall, 1991.

[8] J.M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 1989.

[9] A. Vogel. On ODP's architectural semantics using LOTOS. In J. de Meer, B. Mahr, and O. Spaniol, editors, *2nd International IFIP TC6 Conference on Open Distributed Processing*, pages 340–345, Berlin, Germany, September 1993.