A Declarative Executable Language based on OCL for Specifying the Behavior of Platform-Independent Models

Pierre Kelsen, Elke Pulvermueller, and Christian Glodt

University of Luxembourg
Faculty of Sciences, Technology and Communication
Luxembourg

Abstract. Model-driven architecture aims at describing a system using a platform-independent model in sufficient detail so that the full implementation of the system can be generated from this model and a platform model. This implies that the platform-independent model must describe the static structure as well as the dynamic behavior of the system. We propose a declarative language for describing the behavior of platform-independent models based on a hybrid notation that uses graphical elements as well as textual elements in the form of OCL code snippets. Compared to existing approaches based on action languages it is situated at a higher level of abstraction and, through a clean separation of modifier operations and query operations, simplifies the comprehension of the behavioral aspects of the platform-independent system.

Keywords: model-driven architecture, platform-independent model, action language

1 Introduction

The vision of model-driven architecture - and more generally of model-driven software development - is to enable automatic system generation from models, i.e. to produce software which is automatically constructed by software based on abstract models. Generation is an industrially applied practice nowadays. It eliminates some tedious tasks giving time for the more challenging aspects of the software development process (e.g. the requirements and domain modeling).
Basic approaches underpinning model-driven development may be classified as follows:

– Generation for a limited and/or very specific domain (e.g., embedded systems). These approaches are closely related to product-line development [2]. The domain determines the realization details and generation refers to the configuration of predefined system units.
– Generation of skeletons from higher-level models (often UML models). The skeletons have to be augmented by source code. In general, this is performed manually.
– Generation based on semantic model enrichment. Instead of augmenting the generated output with realization details, this approach adds additional semantics to the

top-level models. OMG's action language semantics [16, 1] and the various notational realizations on top of this standard (e.g., ASL [17], SMALL [13]) count as the most prominent representatives for abstract languages used to describe model details.

In this paper we focus on the semantic model enrichment approach because it holds the promise of full code generation and is applicable to a wide variety of systems. This approach concerns mainly the description of the behavior of a system since UML provides sufficient facilities for structural modeling. Although UML provides some means of expressing the dynamic aspects of a system, these are either incomplete (e.g., sequence diagrams) or apply only to certain types of systems (e.g., systems with a finite number of states) thus restricting their use for full code generation.

One way of supplementing dynamic information is via an Action Language based on Action Semantics [1]. Action languages follow an imperative style: they are reminiscent (although more abstract than) traditional programming languages such as Java or C++. Furthermore, while the action semantics have been standardized by the OMG, the actual concrete syntax is not part of the standardization. This brings about the prospect of a plethora of action languages being used for describing systems, further complicating the task of understanding and sharing the underlying models.

An alternative approach advocated in [10] is to use OCL for expressing the dynamic behavior via pre- and post-conditions of operations at the platform-independent level. Unfortunately the body of the corresponding operation needs to be written in the platform-specific model (using a traditional programming language) to specify the dynamic behavior. Thus this approach does not allow the full specification of the dynamics at the level of the platform-independent model.

In an attempt to overcome some of these shortcomings we have developed a small behavioral modeling language, named EP [8, 9]. This language is based on two main types of elements: *events* and *properties*. Additional related elements and OCL code snippets augment these basic elements in order to provide an executable specification of the system.

The EP language is able to overcome some of the obstacles of the approaches outlined above:

1. The language is situated at a higher level of abstraction than action languages: much of the dynamics of an operation can be expressed using a graphical notation. The code snippets that are left are OCL expressions describing functions without side effects. Because it cleanly separates modifier operations from query operations the EP language is more declarative than action languages.
2. Unlike OCL-based approaches that specify only pre- and post-conditions, an executable description of the dynamic behavior of the system can be expressed in the EP language, thus enabling full code generation.

The remainder of this paper is structured as follows. In the next section we describe the EP language. Section 3 considers the implications of using OCL and section 4 discusses related work. In the final section we present concluding remarks.

2 The EP language

2.1 An Example: FlightFinder

To illustrate the declarative language, we will make use of an application called *FlightFinder* that will be used as a running example. This application would typically be part of a larger system for performing flight reservations. The application allows the user to enter the data for his specific travel request, that is, which city he flies from and which city he flies to, the departure and return dates as well as the number of passengers (adults/children). The user can query the system for all flights that match the entered data. These flights will be presented in a list on a separate screen. The system also allows an administrator to add new airports and new flights.
Even though this example is quite simple it serves well to illustrate the main concepts by not overburdening the description of the underlying diagrams.

2.2 Structural Modeling

Although our main focus is on behavioral modeling, we need to concern ourselves also with structural modeling since structural elements will be reused in the behavioral model. For the structural modeling we make use of standard UML class diagrams with the following modifications:

– We list as operations only query operations, i.e., operations that do not modify state.
– We add a fourth compartment named ”events” . We may think of events as modifying operations whose semantics will be detailed in the behavioral model. We note here that adding named compartments is a facility provided by UML.
– we define initial values of a property using an OCL ”init” constraint - essentially an OCL expression that determines the initial value of the property
– we define the body of a query operation using an OCL ”body” constraint that describes, using an OCL expression, what a query operation returns.

These modifications can be expressed more formally using an UML profile (omitted).
Figure 1 shows the static structure of the FlightFinder application: it comprises 8 classes partitioned into three packages - the main (unnamed) package containing the main business classes, the *ui* package containing classes representing the user interface (three screens) and the *system* package containing a single class *Date*. Note that we have left out the OCL *init* and *body* constraints (which could be attached as notes to the corresponding properties and query operations) in order not to overburden the diagram.

2.3 Behavioral Modeling

For behavioral modeling we depart from UML by introducing a new executable language for modeling the behavior of a system: this language expresses the dynamic behavior of the system by using events and properties (from which we derive the name – EP – of the language) from the class diagram as first-class entities. We shall define the EP language by giving its abstract and concrete syntax as well as its static and dynamic semantics.

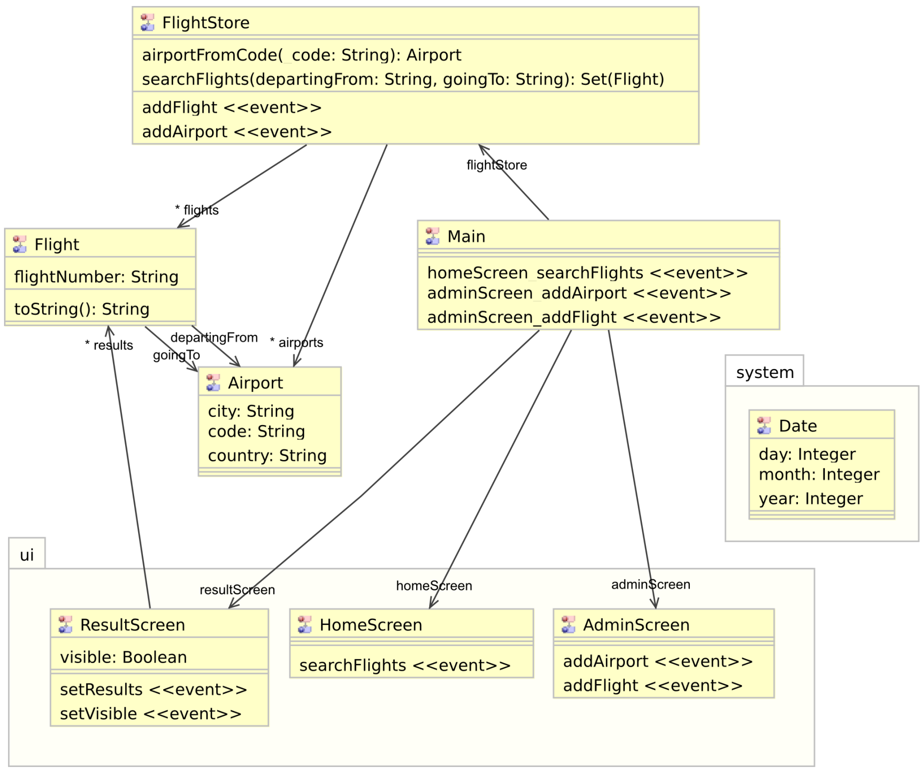


Fig. 1. Class diagram of the FlightFinder application

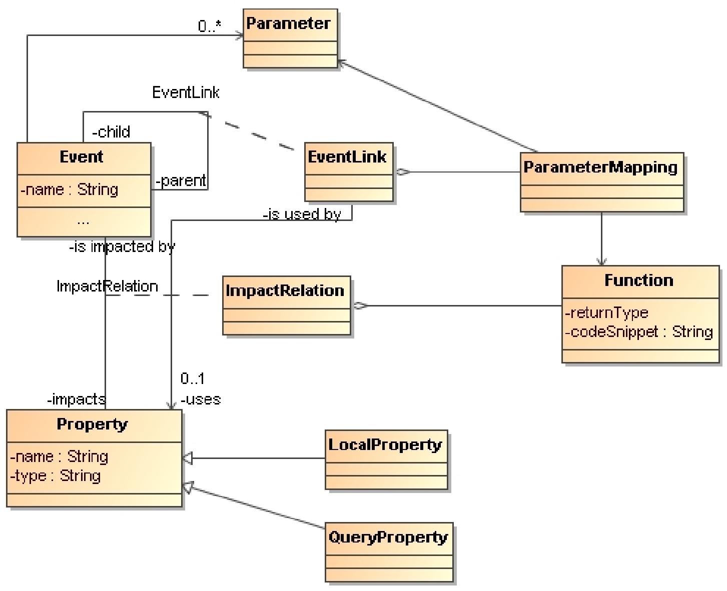


Fig. 2. Behavioral Metamodel

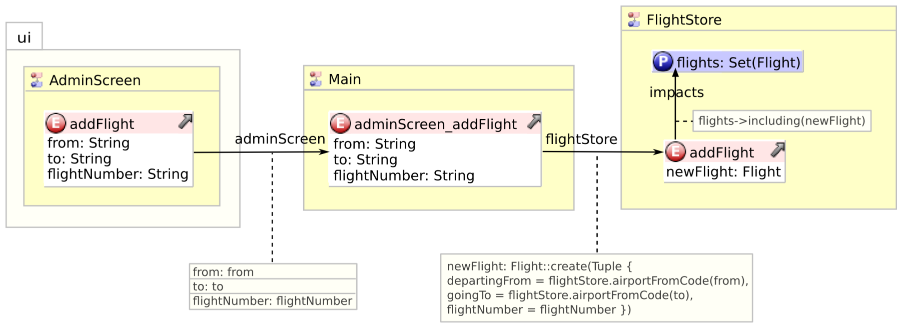
Abstract Syntax In figure 2 we describe the abstract syntax of the language via its UML metamodel. From hereon we shall use the name *EP-model* to denote an instance of this metamodel.
The main entities are events, properties and functions. An event can have parameters, each parameter having a name and a type. An event link connects an event to a child event via a link property and is labeled with a link property which is either a property or a parameterless query operation (denoted by the *QueryProperty* class in the meta-model). Each event link carries one parameter mapping per parameter of the child event. The parameter mapping is an OCL code snippet (represented by a *Function* object) that expresses the value of the parameter of the child event in terms of the parameters of the parent event and in terms of query operations and properties of the model containing the parent event. An event can modify the state of the system by impacting a property in its class: the *impact* link carries an OCL code snippet that expresses the new value of the targeted property.

Concrete Syntax In addition to the abstract syntax expressed by the metamodel we also need to define the concrete syntax of EP-models. We use the following conventions:

– We represent events as boxes with the event name (prefixed by ”E”) at the top and the parameters (name and type) listed below.
– We represent the event link by an arrow from the parent event to the child event labeled by the link property (the property that the link uses) .

– We attach to the event link a note listing the parameter mappings: this is a list of items of the form <parametername>:<code-snippet> where the code snippet is an OCL expression.
– We denote a property by a box containing the name of the property and its type (prefixed by ”P”).
– We denote an *impact* link using an arrow with an attached note containing the code snippet.

In figures 3 and 4 we show partial views of the EP-model of the FlightFinder system: figure 3 includes those events reachable from the *searchFlight* event while figure 4 shows the events reachable from the *addFlight* event. We present these partial views because the full model would be difficult to read.



```
from: from
to: to
flightNumber: flightNumber
```

```
newFlight: Flight::create(Tuple {
departingFrom = flightStore.airportFromCode(from),
goingTo = flightStore.airportFromCode(to),
flightNumber = flightNumber })
```

Fig. 3. View of the EP-model for the *addFlight* event

Static semantics The *static semantics* of an EP-model state the rules that determine whether an EP-model is well-formed. We describe the rules using natural language:

1. Each event link must carry a parameter mapping for each parameter of the target event and each such parameter mapping is an OCL expression returning a value whose type corresponds to the type of the target parameter.
2. The graph induced by the event links on the set of events reachable from a given event is a directed tree rooted at this event, i.e., each node other than the root event has in-degree 1 in this graph and the root has in-degree 0.
   This rule is necessary to ensure that the triggering of an event will not lead to the same event being triggered twice, possibly with different parameter values. It also ensures that the graph induced on the events by the event links is acyclic. If we think of events as modifiers (operations that modify state) and of event links as representing modifier invocations, then this rule prevents an event from leading to an endless loop of modifier invocations.

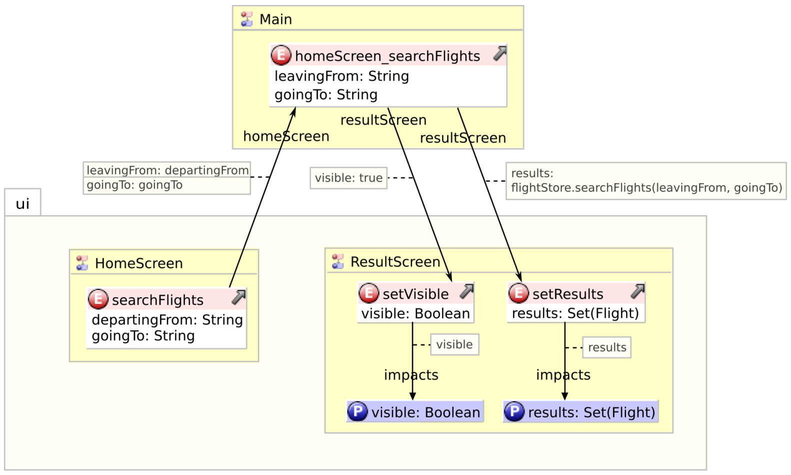


Fig. 4. View of the EP-model for the *searchFlight* event

Dynamic semantics For the dynamic semantics we have to define the meaning of a well-formed EP-model. At run-time the system state comprises a set of *instances*, essentially the object graph complying with the class diagram describing the static structure of the system. We describe the dynamic semantics by defining what happens to the system state when an event is triggered on an instance. Let the *old state* denote the system state just before an event is triggered and let the *new state* stand for the system state right after an event has occurred. The value of a query operation on an instance is determined by computing the value of the OCL expression on the old state. When an event is triggered, all properties impacted by this event are set to the values of the OCL expressions attached to the impact links. For each event link leaving this event we evaluate the link property (with respect to the old state) to determine the target instance referred to by this link property. We also evaluate the value of each parameter of the child event by evaluating the OCL expressions in the corresponding parameter mappings. We then recursively trigger the event on the target instance with the computed arguments.

We illustrate the dynamic semantics using the example of the *searchFlights* event from figure 4 . Suppose that the *searchFlights* event is triggered on an instance of *HomeScreen* with arguments *departingFrom* (code for departure airport) and *goingTo* (code for arrival airport). The event link from *searchFlights* to *homeScreenSearchFlights* implies that the *homeScreenSearchFlights* event will then be triggered on the *Main* instance that refers using its *adminScreen* property to this instance. The value of the arguments of that event are calculated using the OCL expressions attached to the event link: in this case the arguments from *searchFlights* are simply passed on unchanged. The event links from *homeScreenSearchFlights* to *setVisible* and *setResults* in the *ResultScreen* model indicate that these two events are now triggered (in arbitrary order) on the *ResultScreen* instance referred to by the *resultScreen* property of the *Main* instance.

The *setVisible* event is triggered with argument *true* (see code snippet); this event impacts the visible property by setting it to the value of the *visible* parameter, that is, to *true* (see code snippet on the impact link). The *setResults* event is triggered with an argument given as a set of result flights (indicated by the type - Set(Flight) - of the results parameter). The value of this argument is given by invoking the *searchFlights* operation on the *FlightStore* instance referred to by the Main instance via its *flightStore* property (see class diagram).

3 Use of OCL

In this section we discuss the use of OCL within the context of our behavioral modeling language.
Let us first summarize where OCL is used in our modeling approach:

1. At the class diagram level OCL constraints are used to express the initial values of properties and the body of query operations.
2. at the level of the EP-model, OCL expressions are attached to event links and to impact links.

At the structural level we have separated query and modifier operations. While query operations are defined in the structural model via OCL expressions, the definition of the modifier operations (the ”events”) is relegated to the EP-model. In the EP-model OCL expressions are used for defining parameter mappings on event links and new property values on impact links. The benefits of clearly separating query from modifier operations are well recognized [14, 3]. Indeed freely mixing calls to modifier operations and query functions, as is currently done in action languages, makes it difficult to understand the effect an operation has on the system state and thus negatively affects the effort needed to comprehend a system.
In a sense we are using OCL as a functional programming language that includes object navigation facilities. The original intention for OCL is to express constraints on UML models. We believe its use as a ”programming language” is justified in this context by the fact that the code snippets are combined with UML models and EP-models: traditional functional programming languages are not easily adapted for this purpose. Furthermore the abstract nature of OCL, i.e., the fact that it is quite independent of any platform makes it a good candidate for annotating elements of a platform-independent model. Since OCL expressions are side-effect free and modifier operations are clearly separated from the query operations, this behavioral description of the system should be easier to understand in the same way that a functional program is often easier to understand than an object-oriented one.
One drawback of using OCL is the lack of certain features that are taken for granted in more traditional programming languages. We give only two concrete examples: there are no built-in type *Date* or *Time* and there is no operation on the *String* type that tests whether another string is a substring of a string. If we want to use OCL for realistic examples we need to extend it so that we can express the behavior of a large system.
More seriously OCL 2.0 does not seem to be Turing-complete in the sense that OCL expressions only represent primitive recursive functions [12]. As pointed out in [18] the

expressiveness of OCL can be increased by adding recursive operation invocations. We allow this by permitting the OCL expression for a query property with parameters to refer to itself. As explained in [18], however, this results in a Turing-complete expression language, at the cost of not being able to guarantee termination of an expression evaluation.

4 Related Work

We discuss some existing approaches for behavioral modeling of platform-independent models. The main approach advocated by the OMG group for model-driven architecture are action languages that conform to the Action Semantics. The Action Semantics describes the abstract syntax and semantics of action languages but does not propose a concrete syntax. Examples of concrete action languages are the Action Specification Language (ASL) [17, 11], the BridgePoint Action Language (AL) [7], the Kabira Action Semantics (Kabira AS) [6], and the action language subset of the Specification and Description Language (SDL), an international standard widely used in the telecommunication industry [19]. The multitude of different action languages is a first problem we encounter when using action languages.

A more fundamental problem is the intermixing of non-modifying actions and modifying actions. Indeed according to the Action Semantics an action can compute values, navigate and read properties and call query operations but it can also write properties and call modifying operations. In this sense a program written in an action language is similar to one written in a traditional imperative or object-oriented style. This intermingling of modifying and non-modifying actions contributes much to software complexity; a clear separation is a definite argument in favor of our approach. The following code snippet is written in the ASL action language and expresses the setting of the *results* and the *visible* properties in the *ResultScreen*:

```
rs2:setVisible[TRUE] on resultScreen
{theSet} = fs1:searchFlights[leavingFrom,goingTo] on flightStore
rs1:setResults[{theSet}] on resultScreen
```

This example illustrates the mixing of calls to query operations (*searchFlights*) with invocations of modifier operations (*setResults* and *setVisible*). It also shows that the sequential execution is fixed in the action language by the order in which the actions were written down. This is an undesirable feature of imperative languages that is not present in our declarative approach.

We remark that there were attempts to align action languages with the OCL by embedding OCL expressions into new syntax constructs for actions [5]. That result can be seen as an instance of the more general problem of behavioral modeling with OCL, a problem to which we provide a more declarative solution in the present paper.

Action languages are also used at higher levels of abstraction - such as in Kermeta [15] - itself inspired from the UML action language Xion [15]. In Kermeta an action language is used to define the behavior of MOF models. The action language itself is imperative and object-oriented and thus suffers from the same shortcomings as traditional UML-based action languages.

5 Conclusion

In this paper we have presented a declarative language for behavioral modeling of platform-independent models. Existing approaches are mainly based on textual action languages that are imperative in style. The main advantage of our approach is a clear separation of modifier operations and query operations that facilitates the comprehension of the behavior of a platform-independent system.
Unlike traditional action languages our behavioral description language is composed of graphical as well as textual elements, the latter being composed of code snippets representing side-effect free OCL expressions. In this paper we have shown that OCL is well-suited in this context since it is platform-independent and is Turing-complete, provided we allow recursive query calls in expressions. Further work is needed to:

– investigate extensions of OCL for behavioral modeling of realistic systems (in combination with a declarative language such as EP); these extensions will at the least require additional OCL types and operations that are currently lacking
– provide tool support for modeling platform-independent models with the goal of fully automatic code generation. We have developed a first prototype supporting abstract modeling; it is based on the DEMOS tool [4] that supports platform-specific executable modeling
– analyze the scalability of our approach to large software systems; the availability of a suitable tool is a precondition for this investigation
– investigate the application of our behavioral modeling approach to systems in which different aspects are expressed using different domain-specific languages

References

1. Alcatel, I-Logix, Kennedy-Carter, Inc. Kabira Technologies, Inc. Project Technology, Rational Software Corporation, and Telelogic AB. Action semantics for the UML. In *Document ad/2001-03-01. OMG*, 2000.
2. K. Czarnecki and U.W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.
3. E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.
4. Christian Glodt and Pierre Kelsen. Demos: a tool for declarative executable modeling of object-based systems. In *OOPSLA Companion*, pages 716–717, 2006.
5. Stefan Haustein and Jörg Pleumann. OCL as expression language in an action semantics surface language. In Octavian Patrascoiu, editor, *Workshop on OCL and Model Driven Engineering, UML 2004 Conference*, pages 99–113. University of Kent, 2004.
6. Kabira Technologies Inc. Kabira Action Semantics. http://www.kabira.com.
7. Project Technology Inc. BridgePoint Action Language (AL). http://www.projtech.com.
8. Pierre Kelsen. A simple static model for understanding the dynamic behavior of programs. In *12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 46–51, 2004.
9. Pierre Kelsen. A declarative executable model for object-based systems based on functional decomposition. In *ICSOFT (1)*, pages 63–71, 2006. full version availble at `http://lassy.uni.lu/demos/documentation/TR_LASSY_06_06.pdf`.

10. Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
11. Kennedy Carter Ltd. Action Specification Language (ASL). http://www.kc.com.
12. Luis Mandel and María Victoria Cengarle. On the expressive power of OCL. In *Proc. FM'99 – Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France*, volume 1708 of *Lecture Notes in Computer Science*, pages 854–874. spv, 1999.
13. Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture.* Addison-Wesley, Boston, 2004.
14. Bertrand Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
15. Pierre-Alain Muller, Philippe Studer, Frédéric Fondement, and Jean Bézivin. Platform independent Web application modeling and development with Netsilon. *Software and System Modeling*, 4(4):424 – 442, 2005.
16. OMG. OMG Unified Modeling Language Specification (Action Semantics)., January 2002.
17. C. Raistrick, P. Francis, and J. Wright. *Model Driven Architecture with Executable UML.* Cambridge University Press, 2004.
18. Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.
19. International Telecommunication Union. Specification and description language (SDL),Technical Report Z.100, ITU-T, 1999.