An Educational Case Study in Protocol Verification and Distributed Observation

Claude Jard, Thierry Jéron

IRISA,[*]

Campus universitaire de Beaulieu,

F-35042 Rennes, France

E-mail: {jard, jeron}@irisa.fr

Tel: +33 2 99 84 71 93, +33 2 99 84 74 64

Fax: +33 2 99 84 25 32

[*]Claude Jard is member of the CNRS, Thierry Jéron is member of INRIA.

Abstract

This paper presents a case study given to students in computer science during a course on formal techniques for protocol validation. The course intends to teach some basic notions on specification, simulation, verification, testing and distributed observation. The case study is the classical alternating-bit protocol. The exercise first consists in some simple modeling and simulation of the specification and verification of some basic service properties. It is then followed with the problem of the observation of distributed executions of the protocol. This kind of experiment was found to be useful for students since they gain an understanding of the importance and necessity of formal methods for protocol validation.

Key-words : Protocol, Property, Validation, Verification, Model Checking, Distributed observation, Case study.

1 Introduction

Over the two last decades, a lot of research has been devoted to the use of formal techniques for the validation of reactive and concurrent systems. The use of formal techniques is essential for such systems which are often safety critical and/or very complex. Formal methods allow non ambiguitous specifications and early detection of errors with techniques such as simulation, model checking, proof and test generation. Nevertheless, these techniques are only marginally used in industry. There are several reasons for this situation but one of the major ones is probably the lack of education on formal techniques.

This paper focuses on a case study chosen as an illustration of a course on protocol validation. This course is given during the last year of education of students in computer science and engineering. It corresponds roughly to a masters degree in US. The background of students entering this course is poor in formal methods. Among concepts useful for the course, they have acquired some basic notions on automata theory, graph algorithms, propositional logic, concurrency and distributed algorithms. The course on protocol validation is a thirty-hours course separated into two distinct parts. Fifteen hours are under the form of lectures interleaved with fifteen hours of practical work.

The course aims at introducing some basic notions of formal techniques for the validation of protocols. This course starts with an introduction to the model of communicating finite state-machines (CFSM) [Boc78] putting emphasis on the asynchronous nature of the communication. A first approach to formal verification of CFSMs is made using classical notions of language and automata. Some basic notions on Petri nets [Bra83] and their structural analysis are also introduced. We then teach temporal logic and model-checking illustrated with the example of CTL [CES83]. Then follows a lecture about conformance testing and automatic test-case generation. We end with a short review of some techniques for the observation and diagnosis of distributed systems.

This course is illustrated with fifteen hours of experiments on machines, using different tools. Due to time limitations, the experiments only cover simulation, verification and observation. Students are tightly guided during the sessions in order to avoid divergence between different groups.

The specification language used for the experiments is the formal description technique Estelle [ISO89]. The two other standardized formal description techniques LOTOS [LOT85] and SDL [RZ96] could have been chosen as well. One of the main reasons for our choice (made several years ago) has been the availability of some tools for this language. We will come back to this in the conclusion below because things have changed. The aim of the course is not to learn a specific language, but the concepts behind formal validation. So we give a very short introduction to Estelle during the first session. However, students do not have to write complete specifications by themselves. Instead, they are given some initial specifications and have to complete or modify them with simple constructs of the language. In this context, one of the advantages of using Estelle is its simple syntax and its intuitive semantics.

Several tools are used for protocol validation in the case study. Two of them support Estelle:

- **Veda** from Verilog is a commercial tool which can be used for the simulation and verification of Estelle specifications. Veda is not supported anymore, as it is now replaced by the tool ObjectGeode [ALHH93] for SDL specifications [RZ96].

- **Echidna** is a prototype developed by our group [JJ89]. This tool compiles Estelle specifications into executable code that can be executed on parallel machines and a network of Unix stations.

We also use other prototypes developed in our group which are not specific to Estelle. **Viscope** [JJ94], allows visualization in 2D or 3D state graphs, in particular those generated by Veda. **Xhasse**, builds and draws the Hasse diagram of the causal ordering of a distributed observation (the graph of interactions) and **Xbourgeon**, builds and draws the lattice of ideals of the causal ordering (the graph of global states). We will detail all these tools in section 3

Several small protocols are used along the experiments. Among these, the case study on the alternating-bit protocol covers six hours. The first three hours are spent on verification, and the last three hours are spent on distributed observation. The choice of the alternating-bit protocol is not original, but there are several reasons for this choice. First, it is a simple protocol and this is an advantage for students not very skilled in protocols. It is a simple example of the classical problem of distributed agreement, and this makes a link with another course on distributed algorithms. Despite its simplicity, the different variants of the protocol exhibit some interesting behaviors, especially from the observation point of view. Although this is not the main topic of the course, the modeling aspects are also interesting.

This paper is organized as follows. Section 2 presents the principles of the alternating-bit protocol and its specification in Estelle. Section 3 presents the tools used by students. The two following sections describe how these tools are used for simulation and verification (section 4) and for distributed observation (section 5). Section 6 concludes by analyzing the difficulties encountered by the students and what we believe that they have learned. We also investigate some possible future evolutions of the course.

2 The Alternating Bit Protocol and its Specification

2.1 Principle of the Protocol

The alternating-bit protocol is a classic example of a communication protocol. Its aim is to transfer data between two users over a communication medium. The communication is asynchronous using FIFO channels. During the experiment we will consider several properties of the communication medium. We first consider a perfect medium. But we also consider channels that can lose data (we suppose that a lower-level error detection/correction mechanism eliminates messages with unrecoverable errors) or can alter the control bit carried by messages (we assume that the error detection/correction mechanism is not perfect and that there are some residual errors). The protocol uses a control bit in order to detect the "good" sequencing of messages. The bit is used by protocol entities in order to detect the need for a retransmission.

A simple description of the architecture of the protocol and its behavior is given in Figure 1. The behavior is described with extended automata. Each transition is possibly guarded with a message reception and a boolean condition. A transition contains an action consisting of message outputs and modifications of variables. Initially, students are given an incomplete specification with a perfect medium. They have to modify the medium and complete the specification. Thus dashed arrows in Figure 1 represent transitions that have to be added by students.

The specification is composed of five processes. The lower process describes the *medium* which is used by two protocol entities, a *sender* and a *receiver*. Each of these processes communicates with a user, namely *sender_user* and *receiver_user* which model the environment. All communication between processes is asynchronous through FIFO channels.

The *medium* has three states and is initially in state *begin*. Upon reception of a message *mess* from *sender* (respectively an acknowledgment *ack* from *receiver*), it moves to *del_data* (resp. *del_ack*) and records the parameters. In this state, the medium goes back to state *begin* with a non-deterministic choice between three possibilities:

- it sends *mess* transparently to *receiver* (resp. *ack* to *sender*),
- it makes a silent move which models the loss of *mess* (resp. *ack*),
- it sends *mess* to *receiver* (resp. *ack* to *sender*) with an opposite control bit (1-bit). This models a possible message corruption.

The *sender* process has two states. It is initially in state *idle* with its *bit_buf* variable set to 0. From there it moves to state *wait_ack* upon reception of a *data_req* from its *sender_user*, registers the *data* carried by the message in *data_buf* and sends a *mess* to the medium carrying *data_buf* and *bit_buf*. In *wait_ack* it goes back to *idle* upon reception of *ack* carrying the right control bit (bit_buf = bit), it changes the control bit (bit_buf = 1 - bit_buf) and sends a *confirm* to *sender_user*. If the control bit is not correct (a corruption is detected), it sends again the message *mess* with the registered data *data_buf* and bit *bit_buf* and it stays in state *wait_ack*. If the timeout *retrans_time* expires, the same action is performed in order to resist against message loss.

The *sender_user* has two states. Initially in state *idle*, it may spontaneously move to state *wait* while sending a *data_req* to *sender*. It goes back to *idle* upon reception of a *confirm*.

The *receiver* process has only one state in which it waits for a message *mess* from the medium. Initially its variable *bit_buf* is set to 0. If the received *mess* carries the right control bit (bit = bit_buf), it sends a data indication *data_ind* to its user *receiver_user* and an acknowledgment *ack* to the medium with the same control bit, and changes the control bit to its opposite value. Otherwise it only sends back *ack* with the received bit.

Finally, the *receiver_user* has a unique state in which it waits for a data indication *data_ind*.

2.2 A Short Description of the Estelle Language

Estelle is a formal description technique standardized by ISO [ISO89]. It is dedicated to the specification of communication protocols and distributed algorithms. Estelle is based on the underlying model of communicating finite-state machines extended with Pascal language constructions. An Estelle specification is described by a hierarchy of communicating processes. A process is an instance of an interface (*module*) associated with a behavior (*body*). The interface defines the interaction points (*ip*) and an attribute (*systemprocess, systemactivity, process, activity*). A process can be dynamically created, connected or deleted. Communication between processes is made by message exchange through bi-directional, unbounded FIFO channels connected to interaction points, and by export of variables from a "son" to a "father" process. Several behaviors can be associated with the same interface. The behavior of a process is specified by an automaton extended with Pascal-like constructions and communication features. A transition is composed of a guard and a possibly labeled action. A guard is composed of a starting state, an input (that is accepted only if the message is at the head of the FIFO queue) and a condition (a boolean condition on local variables or input message parameters). "Spontaneous" transitions (without input clause) can be used to model timers. When a transition guarded by *delay(val_min,* *val_max)* becomes fireable, one must wait at least *val_min* time units and at most *val_max* time units before firing it. *Priority* clauses can be used to define an order on fireable transitions. The choice of which transition to be fired among those that are fireable in the process (whose guards evaluate to true) is non-deterministic. The action is composed of a state transfer and a pseudo-Pascal program extended with some specific constructs, like the *outputs* through interaction points.

The tools Veda and Echidna consider only a subset of the Estelle language. In particular they do not allow dynamic creation and destruction of processes. In Echidna, attributes of modules play no role. The effective architecture will define the semantics: interleaving between processes placed on the same physical process and full asynchronous parallelism between different physical processes. Echidna considers additional predefined functions for tracing (like Pascal *write*) and timing (returns local time of the processor in seconds).

2.3 Estelle Specifications of the Alternating Bit Protocol

In the course under consideration, Estelle specification of the alternating-bit protocol is provided to the students. The specification consists of (cf. Figure 1) two end-users, two protocol entities, a sender, a receiver, and a communication medium. The initially given specification is incomplete because it considers a perfect communicating medium that correctly delivers every message and acknowledgment. For the case study, students have to modify and complete the specification. The goal is to see how the protocol can be adapted (if possible) to the loss and corruption of messages by the medium. So they have to add the possibility for the medium to lose messages and acknowledgments, and to corrupt messages and acknowledgments. These added transitions correspond to dashed arrows in the medium description. As the initial specification does not consider the possibility of losing messages and acknowledgments, protocol entities do not try to detect these possibilities. Therefore, students have to add a timer mechanism for retransmission (dashed arrow in the sender entity). The initial Estelle specification is 300 lines long.

3 Techniques and Tools

During the case study, students have to use several tools dealing with Estelle specifications or intermediate objects produced. One of these tool, namely Veda, is commercial. The others have all been developed in our research group. We now describe these tools in more detail.

3.1 Veda

Veda (produced by the company named Verilog) is a commercial tool for the analysis of Estelle specifications using simulation and verification. The analysis can be done using *observers* that describe some desired properties of the specification. Before simulation and verification, the Estelle specification and the associated observers must be compiled into simulation code. A graphical interface then gives access to different features:

Step by step simulation: In this case, the specification is first placed in its initial global state. Fireable transitions are proposed and the user can choose one of them, which results in a new global state. The same process is iterated from this state. The user can undo transitions, save scenarios and load scenarios. He/she can also view the evolution of some aspects of processes such as variables, FIFO queues, control states, etc. The simulation may also be used for verifying properties on the explored sequences using an observer. Observers are described below.

Random simulation: Rather than choosing fireable transitions, the user can let the system choose randomly among fireable transitions. In this mode, the user can limit the number of explored states in the sequence.

Exhaustive simulation: In this case, Veda explores all possible behaviors of the specification. The exploration can be done by a breadth-first or depth-first exploration. Some parameters such as a depth limit, maximum number of states, and stop conditions can limit the exploration. The exploration can also be done starting from a state reached through an initial scenario. The explored state graph (global states and/or transitions) can be saved into a file. If no observer is specified, the exhaustive simulation detects deadlocks and can save scenarios leading to these deadlocks.

Verification by an observer: An observer is essentially an extended automaton given in a syntax quite similar to Estelle. It specifies some states, among which some are labeled “error” or “success”, depending on the property being checked. The observer defines probes in order to access the objects of the specification. Probes can be defined for process instances, allowing observation of the evolution of their variables and queues and the firing of transitions. The transitions of the automaton are guarded by names of transitions of the specification. A transition of the observer is executed if the corresponding transition of the specification is fired. This implements the concept of ideal observer knowing “instantaneously” the global state of the specification. This imposes the restriction that observers are deterministic, since if not, a dynamic error may occur. Several exploration options can be set in order to speed-up the verification. This allows on-the-fly verification [FJJM93] to be undertaken during a depth-first traversal, by stopping the exploration as soon as an error is found. Scenarios leading to error or success states can be saved in files.

3.2 Viscope

Viscope [JJ94] is a prototype tool which allows state-graphs to be drawn in two or three dimensions. The tool accepts state graphs in the format generated by Veda. The two-dimensional representation is simply done by a breadth-first search, drawing states according to their depth in the graph. This is also the case in three dimensions but in this case information on concurrent transitions is preserved by representing inter-leavings by "diamonds". In order to help the understanding of behaviors, transitions are colored according to the process in which they occur. This representation ensures that two distinct states will have different coordinates in 3D. Viscope generates a Postscript file containing several "conical" 2D-projections of the 3D-representations.

3.3 Echidna

Echidna [JJ89] is a prototype tool for the experimentation of distributed algorithms on distributed architectures. The input language is Estelle. Several target architectures can be used, such as networks of Unix stations, parallel machines (the hypercube iPSC/2), machines based on transputers, etc. Echidna also implements a global time service to precisely date the observed events.

Echidna compiles an Estelle specification into a C program, and links this C program with predefined libraries, some of them target architecture dependent libraries. The object program can then be executed on the target architecture. For example, on a network of Unix stations, a Unix shell is opened on each machine before execution. The connected machines are declared by a specific command. Then the program is launched on each machine. By default, logical processes are attributed a physical station by a round-robin policy modulo the number of physical stations. An attribute can also be added to Estelle processes in order to define a logical architecture. Some execution options allow definition of an execution duration, a seed for the random generator that is used for the computation of fired transitions and a trace of each interaction that allows to analyze message exchanges.

3.4 Causal, Xhasse and Xbourgeon

Causal, **Xhasse** and **Xbourgeon** are prototypes used for the analysis of distributed execution traces in conjunction with Echidna. These prototypes suppose that we have observed a trace of a distributed execution augmented with vector time-stamps [Fid88, Mat89]. These time-stamps define a causal ordering $\prec$ of observed events defined as follows: Let $\{P_1, \ldots, P_n\}$ be a finite set of processes, and let $E = X \uplus \bar{X} \uplus O$ be the finite set of events occurring during the execution ($\uplus$ denotes the disjoint union) X contains all the sending events (unobservable in principle), $\bar{X}$ contains the corresponding receptions (for the sake of simplicity, communication channels are considered point-to-point) and O is the set of observable events. In this general framework, observable events can be any kind of events. One can consider that E is split into the disjoint subsets E_i of events that are local to P_i: $E = \biguplus_{1 \leq i \leq n} E_i$. The relation $\prec_i$ is the sequential ordering of events appearing on process P_i. The causal relationship [Lam78] in E^2 is the smallest relation $\prec$ satisfying :

1. $\forall i \in \{1..n\}, \forall x, y \in E_i, \quad x \prec_i y \Longrightarrow x \prec y$
2. $\forall x \in X, \quad x \prec \bar{x}$
3. $\prec$ is transitive.

For observation purpose, we only consider the sub-order $\widetilde{O}$ induced by the set O of observable events partially ordered by the relation $\prec$. In practice, a trace is a set of observable events labeled with their names, their process names and their vector time-stamps.

Causal is a preprocessing program which reorders these events into a linear extension of the causal ordering i.e., a sequence where the causal ordering is preserved: if $x \prec y$ then x appears before y in the sequence.

Xhasse takes as input this linear extension and computes and draws the Hasse diagram of the causal ordering. The Hasse diagram is the transitive reduction of the ordering that places events issued by the same process on a vertical line.

Xbourgeon includes a part of **Xhasse**. It takes the linear extension, computes the causal ordering, computes and draws the lattice of ideals of the causal ordering $\widetilde{O}$. An *ideal* A of a poset $\widetilde{P}$ is a subset closed by precedence: $\forall x \in A, \forall y \in P,\ y \leq_{\widetilde{P}} x \Rightarrow y \in A$. The set of ideals $I(P)$ of a poset $\widetilde{P}$, ordered by set inclusion, forms a distributive lattice $\widetilde{I(P)}$ [Bir67]. Intuitively, each sequence from the bottom to the top of the lattice is a possible linear extension of the causal ordering. This means that the lattice represents all possible executions compatible with the observed trace. This lattice can also be used for the verification of some properties on distributed executions [JJJR94].

4 Protocol Verification

The first three hour session of the experiment is devoted to verification. In our course the purpose is twofold. Students have to familiarize themselves with verification techniques. This is achieved by using the Veda tool and its main functionalities, namely simulation, reachability analysis and model checking. Students also have to understand the problem of distributed agreement which is illustrated by the alternating-bit protocol. This is achieved by using different versions of the protocol and by varying the properties of the communication medium in an incremental way.

4.1 Simulation and State Graph Generation

The initial step of the case study consists in understanding the behavior of the protocol. Students are given an initial Estelle specification of the alternating-bit protocol without loss and corruption of messages. The students have to represent the architecture of the protocol and to detail the behavior of each process by extended communicating automata. This "reverse-engineering" phase leads them to represent the protocol by something close to Figure 1. This is not sufficient for a deep understanding of the global behavior of the protocol. Thus they use the Veda simulator (in step-by-step mode using the **watch** facility) in order to observe the evolution of control states, variables and queues (cf. Figure 2).

Then they launch a state graph generation using the exhaustive simulator of Veda. Students notice that no deadlock is found in this version of the protocol. The graph is saved in a file and visualized with Viscope (cf. Figure 3). This helps in understanding the global behavior of the protocol.

4.2 Property Checking

We then give an observer pattern to the students. The observer only prints messages on the occurrence of service events, i.e., outputs of *data_req* and inputs of *confirm* by the sender user and outputs of *data_ind* by the receiver protocol entity. Students are then asked to complete this observer in order to really specify the service property, i.e., the sequencing of $!data\_req$, $?confirm$ and $!data\_ind$. This should result in the automaton represented in Figure 4 in which an error state is reached if the above mentioned sequencing property is not respected.

In Veda, the observer can be simplified as it implicitly loops in each state on the occurrence of unspecified outgoing transitions. The actions labeling transitions are not exactly those described in the figure. They consist of the labels naming the atomic transitions of the Estelle specification which contain the considered interactions. By linking the observer with the specification and by using Veda for exhaustive simulation, it is possible to prove that the protocol is deadlock free and correct with respect to the given property. The correctness with respect to the service property is evident, as there is no loss or message corruption. The state graph generated during the verification can also be viewed using Viscope. This graph is the synchronous product of the state graph of the protocol with the observer. This explains why it may be unfolded compared to the state graph.

Next, students have to modify the communication medium in order to add the possibility of randomly losing messages and acknowledgments. This modeling is simply done by adding in the medium spontaneous transitions from states *del_data* and *del_ack* that do "nothing". According to the Estelle semantics, any fireable transition can be fired. Therefore, the added transitions simulate a random loss. Using the Veda simulator, we can now analyze the new protocol and detect a deadlock. This situation occurs because no retransmission of message was anticipated in the sender entity. Students are then asked to add the retransmission using a timeout modeled by an Estelle *delay* clause. A new verification with Veda can now prove that there is no deadlock, and now the protocol is correct in presence of loss of messages and acknowledgments.

Next we assume that the medium can corrupt messages. The modeling is done in Estelle by adding a new transition from *del_data* to *begin*, which transmits the message with an incorrect bit. Again, the Estelle semantics models the random choice. Verification with Veda still proves that the protocol resists the erroneous medium, i.e. it is correct despite the medium.

But when we add the possibility of corrupt acknowledgments, Veda detects a sequencing error. Students are then asked if it is possible to correct the protocol. As it is an example of a distributed agreement, there is no solution in the presence of loss and corruption in both directions. We explain this problem by the classical image of "Byzantine armies": a "red" army is split in two ("sender" and "receiver") by a "blue" army (medium). The "red" army wants to coordinate its attack using messengers in both directions (messages and acknowledgments). Coordination takes place when after sending a request, the sender receives a confirmation and the receiver receives an indication. If messengers are captured, new ones are sent, that is this coordination resists the capture of messengers. But if messengers are corrupted and replaced by "spies", the two halves of the "red" army cannot coordinate.

5 Distributed Observation

The aim of this part of the case study is to illustrate some parts of the course on distributed observation. The alternating-bit protocol serves as an example because it is simple and has already been studied and (hopefully) well understood in the preceding phase.

Here students are placed in a new situation. They can now execute the protocol in a real distributed environment, consisting of the machines in the lab connected by a network. The advantage of using our prototype Echidna is that the underlying network is "hidden", so students do not have to implement all transmission mechanisms by themselves. Thus they can concentrate their efforts on understanding the problem of the observation of a distributed execution.

The specification used during this case study considers a medium that can lose and corrupt messages and acknowledgments. Initially, the sender entity is not equipped with a retransmission mechanism. This has to be specified by students. Echidna does not allow the *delay* clause, but gives access to a time primitive, giving the local time of the physical site. Therefore students must implement the retransmission by an "active waiting": a timer variable is set to the local time plus a delay in each transition of the sender where a message is sent. In state $wait\_ack$, a spontaneous transition is added, which is always fireable in this state, but sends a message only when the local time has exceeded the delay. The timeout delay is reset at each output of the message *mess*.

5.1 Distributed Execution

In a first phase, we only ask the students to execute the protocol on the network using Echidna. Using integer attributes of processes interpreted by the Echidna compiler, they can place the *sender_user* and *sender* on the same logical machine, the *receiver* and *receiver_user* on a second machine and the medium on a third one. If they allocate three Unix machines there will be a direct mapping of these three logical machines on the three physical ones. In our case, these machines communicate via TCP/IP. After compilation, s-tudents open one shell for each machine and load the executable code. They can see the execution of the code on each machine through message interactions (cf. Figure 5). They are encouraged to understand a posteriori what has happened. They can notice that analyzing the execution is not easy, even in this situation where all interactions (except parameters) are observed. The goal is to place them in the situation of a global observer which has to build global information from local ones. In general this is possible here because all interactions are observed, the code is known, and the protocol exhibits a very limited amount of concurrency. Nevertheless, they start to understand the problem.

5.2 Global Observer Process

In the next step the students are asked to describe a global observer process in the Estelle specification. The observer is a supplementary process of the specification which runs during the execution of the protocol and collects traces of the distributed execution. These traces are built from events sent by the protocol processes. Notice that the principle is slightly different from the notion of observer in Veda. In this latter case the observer is external to the specification and serves for verification purpose.

The observer process receives service messages from the two protocol entities. This means that the sender sends a message to the observer, indicating that it has received a *data request* or that it has sent a *confirmation*. Similarly, the receiver sends a message to the observer when it sends a *data indication*. In this phase the observer only traces the reception of the messages using a *trace* primitive of Echidna. When students trace all interactions, they are often confused because they realize that the order in which the observer receives and traces the messages is not consistent with the order in which the service events occurred. The goal of the experiment is to convince them that such an observer is unable to analyze the trace of a distributed execution. This requires more information on the causal ordering, which can be provided by logical time-stamps.

5.3 Timestamps and Causal Ordering

In the last phase of the experiment, students are asked to implement logical vector timestamps in the protocol [Fid88, Mat89]. The principle is the following. Each process P_i manages a vector Δ_i of size n, where n is the number of processes. The j^{th} coordinate $\Delta_i[j]$ contains the number of observable events that have happened on P_j, which P_i is aware of. Initially all coordinates of this vector are null. When an observable event occurs on P_i, P_i increments its i^{th} coordinate and sends the event labeled with Δ_i to the observer. When P_i sends a message to P_j, Δ_i is "piggy-backed" to the message. When P_j receives a message from P_i, Δ_j becomes the maximum of Δ_j and Δ_i (coordinate by coordinate). This allows P_j to update its knowledge of the system's past behavior (observable events that have occurred) with the knowledge of P_i. The natural ordering on vectors applied on time-stamps precisely encodes the causal ordering of observable events (in our case, service messages).

In our particular case, the implementation of time-stamps can be significantly simplified because observable events only occur on the two protocol entities sender and receiver.

Thus, vector time-stamps have only two coordinates. The two user entities do not participate in the evolution of timestamps, and the medium only has to convey messages that carry vector time-stamps. Figure 6 represents a possible execution of the protocol with loss and corruption of messages and the evolution of timestamps. In this example, it can be seen that the protocol, in the first phase, copes with the loss of messages. In a second phase, corruption in both directions may lead to a wrong confirmation. Combined with message collision, this may lead to concurrency of service events. This induces a possibility of disordering that violates the service property.

When vector time-stamps are implemented, the observer traces events using the name of their issuing process and the vector time-stamps constructed during the distributed execution. By bounding the execution time, a finite trace can be collected. This trace is reordered in a linear extension using **Causal**. **Xhasse** then builds and visualizes the Hasse diagram of the causal ordering (cf. Figure 7).

Finally, **Xbourgeon** builds the lattice of ideals of the ordering (cf. Figure 8), which gives all execution sequences compatible with the observed trace. It is then possible to check the service property (described in Figure 4) on this trace. As the given specification allows loss and corruption of messages and acknowledgments, it is possible to identify executions which do not satisfy the service property. This does not always work: due to non-determinism in the transitions of the medium, some traces may be correct. More interesting is the case when a trace collected by the observer appears to be correct without consideration of its time-stamps, but is actually incorrect when time-stamps are considered. This situation is not easy to observe since we have a limited control on the distributed execution.

6 Conclusion

6.1 What do the Students Learn ?

At the beginning of the course, our students are almost unaware of formal description techniques, verification methods and trace analysis (it is sometimes even difficult to make them understand the difference between a program and a model). During the course they are faced with new concepts. They sometimes evaluate them as very theoretical notions, and we feel that they are rather skeptical about the possibility of using these notions in a real industrial context. For example, this is the case with property verification or error detection using model-checking and trace analysis. Experiments using cases studies encourages them because they see it "with their own eyes". The main lesson is that it is difficult to understand the behavior of a protocol without the help of formal tools. Even for a simple protocol such as the alternating-bit protocol, they understand that automatic tools are necessary in order to detect errors. The aim of the course is not to present all possible techniques that can be used but only a subset of them. Some of the students will perhaps be faced with the kind of problems encountered during the course. We expect that they will remember that techniques exist and hopefully that they will convince more people about their usefulness.

The evaluation of the understanding of students is done throughout the case study and by an examination at the end of the year. We check that the basic concepts of the course have been understood. We also insert some questions related to the case studies in the examination.

For several years, students have been asked to evaluate their courses and to summarize their reflections to the teachers. For this course on protocol validation, their opinion has always been good. They are especially pleased that lectures are illustrated with case studies and they would even want more.

6.2 Future Evolutions

The case study presented in this paper has been used for several years in different contexts. Some changes and improvements have been made over time depending on research advances and our personal research interests.

But we are now faced with several problems for the future evolution of the course.

First we always have to evolve and incorporate new concepts in the course and, if possible, illustrate them in the experiments. This is the case for test generation which is already a significant part of the course but which for the moment does not appear in the case studies. This is not technically difficult, and we have developed a prototype tool TGV [FJJV96] for this purpose.

A more important problem is the evolution of tools and specification languages. In fact neither Veda from Verilog nor our prototype tool Echidna are maintained anymore. For Veda, the main reason is that Estelle is almost not used in industry. For Echidna, the reason is that we do not work with Estelle in our research anymore. We thus have to choose a new specification language and associated tools.

We have to consider several criteria in our choice. The language should be formal and adapted to protocol specification. It should be easy to learn because we do not want to put stress on the language aspects. It should be supported by tools with some minimal functionalities such as simulation, model checking, code generation, test generation. Also, as we teach to future engineers, usage in industry should be considered. Finally, in our research, we use some specific languages, and thus prefer to invest in one of these languages for teaching. In fact we are helped with PhD students of the group which have to be quickly operational.

LOTOS could be a candidate because of its clear formal semantics and because it is supported by numerous tools of high quality. We even have a connection of our test generation tool TGV with the CADP toolbox [FGK+96]. This toolbox incorporates a LOTOS compiler and gives access to model checking with temporal logic and different bisimulations. However, despite its advantages, LOTOS is very marginally used in an industrial context. Another problem is that we are not using LOTOS ourselves in our research. Finally, we want to put stress on the asynchronous nature of communication while communication in LOTOS is by rendez-vous.

SDL will probably be our choice because it meets almost all our requirements. A first advantage is its apparent simplicity mostly due to its graphical representation. Another advantage of SDL is that it is now widely used in industry in several domains. SDL is supported by several commercial tools including ObjectGeode from Verilog (with whom we have a long experience of collaboration). ObjectGeode functionalities include simulation, model checking and code generation. Our prototype tool TGV is connected to ObjectGeode and an industrial version of TGV has been developed inside ObjectGeode. Thus test generation is also supported: one interesting way would be to play tests against implementations produced by code generation from modified specifications, or possibly we have already investigated. Another possibility is to co-simulate generated test cases with modified specifications, although this possibility does not yet exist in ObjectGeode. Our main problem concerns observation of distributed executions. It will be more difficult to gain the transparency provided by Echidna if we use the code generator of ObjectGeode. However, we can investigate studying distributed observation in a simulation mode. Finally, another advantage of ObjectGeode is that it also gives access to Message Sequence Charts (MSCs) which allow us to describe use-cases.

Another problem concerns the possible change of our case studies. We have had a good experience with the alternating-bit protocol which has a lot of interesting properties for our purpose. In fact the main difficulty is to find a single example which allows us to illustrate most of the concepts seen in the course. Taking a unique illustrating example has the advantage of factorizing the efforts for understanding the example and concentration on the validation concepts. It seems however that the alternating-bit protocol is too simple for the purpose of test generation. Thus perhaps we have to think about a more complex example.

Acknowledgments:

The authors wish to thank Vlad Rusu for his careful reading of the article.

They also thank the referees for their helpful comments.

References

[ALHH93] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON project : A VALidatiON Environment For SDL/MSC Descriptions. In *6th SDL Forum, Darmstadt*, 1993.

[Bir67] G. Birkhoff. Lattice theory. In *American Mathematica Society Colloquium Publications, Vol. XXV*, 1967.

[Boc78] G.V. Bochmann. Finite-state description of communication protocols. *Computer Networks*, 2, October 1978.

[Bra83] G.W. Brams. *Réseaux de Petri: théorie et pratique.* Masson, 1983. Tome 1, 184 p.

[CES83] E.M. Clarke., E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification : a practical approach. *ACM SIGACT-SIGOPS, Symp. Principles of Distributed Computing*, pages 117–126, 1983.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference* *on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*, pages 437–440. Springer Verlag, August 1996.

[Fid88] J. Fidge. Time-stamps in message passing systems that preserve the partial ordering. In *Proc.* 11$^{\text{th}}$ *Australian Computer Science Conference*, pages 55–66, February 1988.

[FJJM93] J.C. Fernandez, C. Jard, T. Jéron, and L. Mounier. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1:251–273, 1993.

[FJJV96] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using On-the-fly Verification Techniques for the Generation of Test Suites. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification, CAV'96, (New Brunswick, New Jersey, USA)*. LNCS 1102 Springer Verlag, aug 1996.

[ISO89] ISO. *Estelle: a formal description technique based on an extended state transition model.* ISO 9074 TC97/SC21/WG6.1, 1989.

[JJ89] C. Jard and J.-M. Jézéquel. A multi-processor Estelle to C compiler to experiment distributed algorithms on parallel machines.

In *Proc. of the 9th IFIP International Workshop on Protocol Specification, Testing and Verification, University of Twente, The Netherlands*. North Holland, 1989.

[JJ94] T. Jéron and C. Jard. 3D-layout of reachability graphs of communicating processes. In *Graph Drawing'94, DIMACS Workshop*, pages 25–33, Princeton, New-Jersey, Octobre 1994. LNCS n^o 894. Paru en rapport de recherche bilingue français-anglais, Irisa n^o 852 et Inria n^o 2334.

[JJJR94] C. Jard, T. Jéron, G.-V. Jourdan, and J.-X. Rampon. A General Approach to Trace-Checking in Distributed Computing Systems. In 14th *International Conference on Distributed Computing Systems, Poznan, Pologne*, pages 396–403. IEEE Computer Society Press, June 1994.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LOT85] LOTOS, a formal description technique based on the temporal ordering of observational behaviour, ISO/ DP 8807, March 1985.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *Proc. Int. Workshop on Parallel and Distributed Algorithms Bonas, France, Oct. 1988*. North Holland, 1989.

[RZ96] ITU-T Recommendation Z-100. Specification and Description Language, 1996.

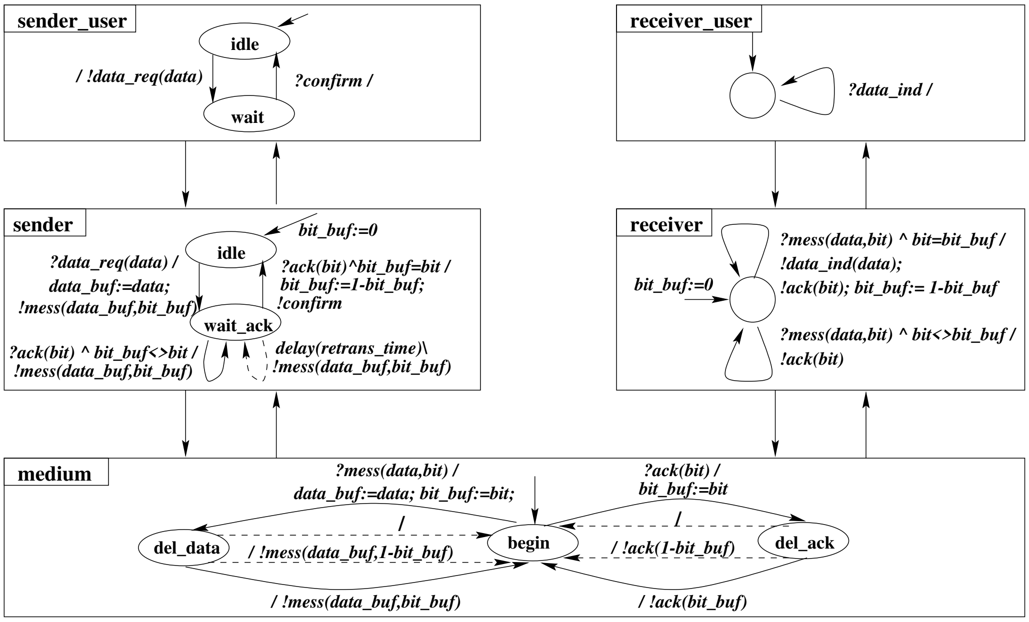

Figure 1: The alternating-bit protocol: architecture and behavior

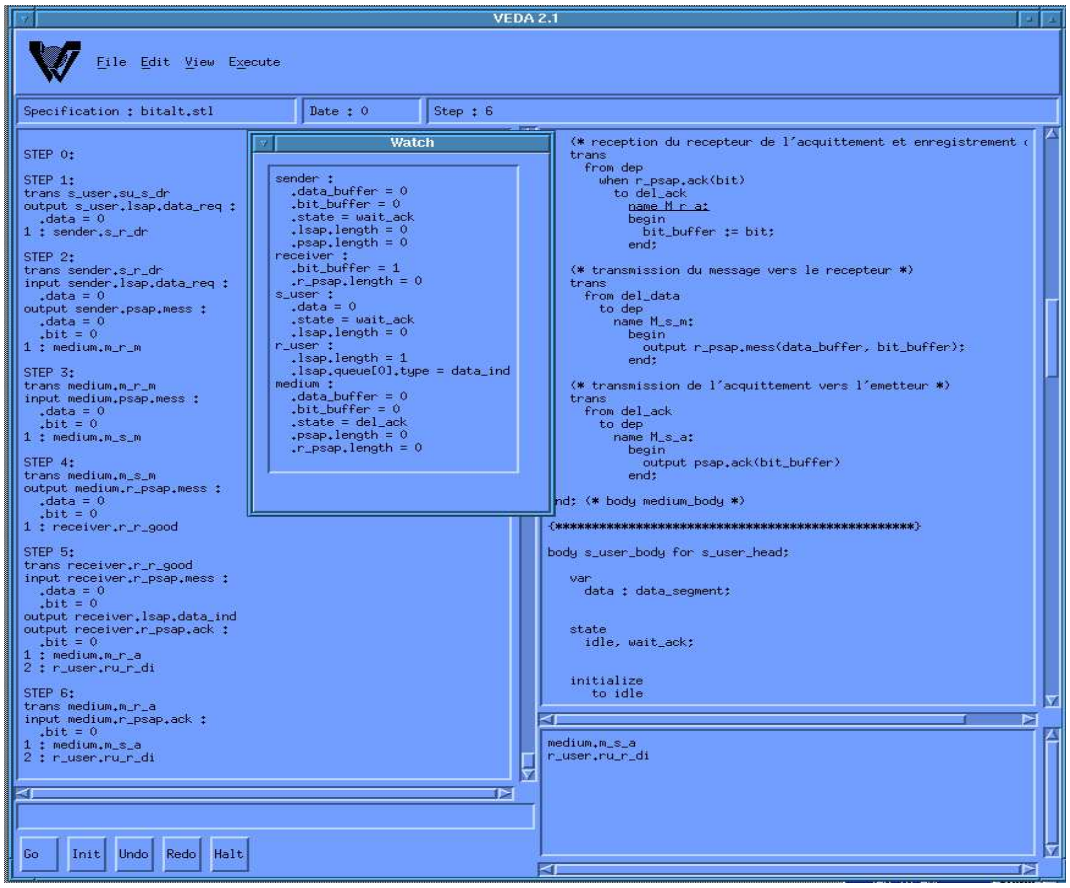

Figure 2: Graphical interface of Veda in step-by-step simulation mode

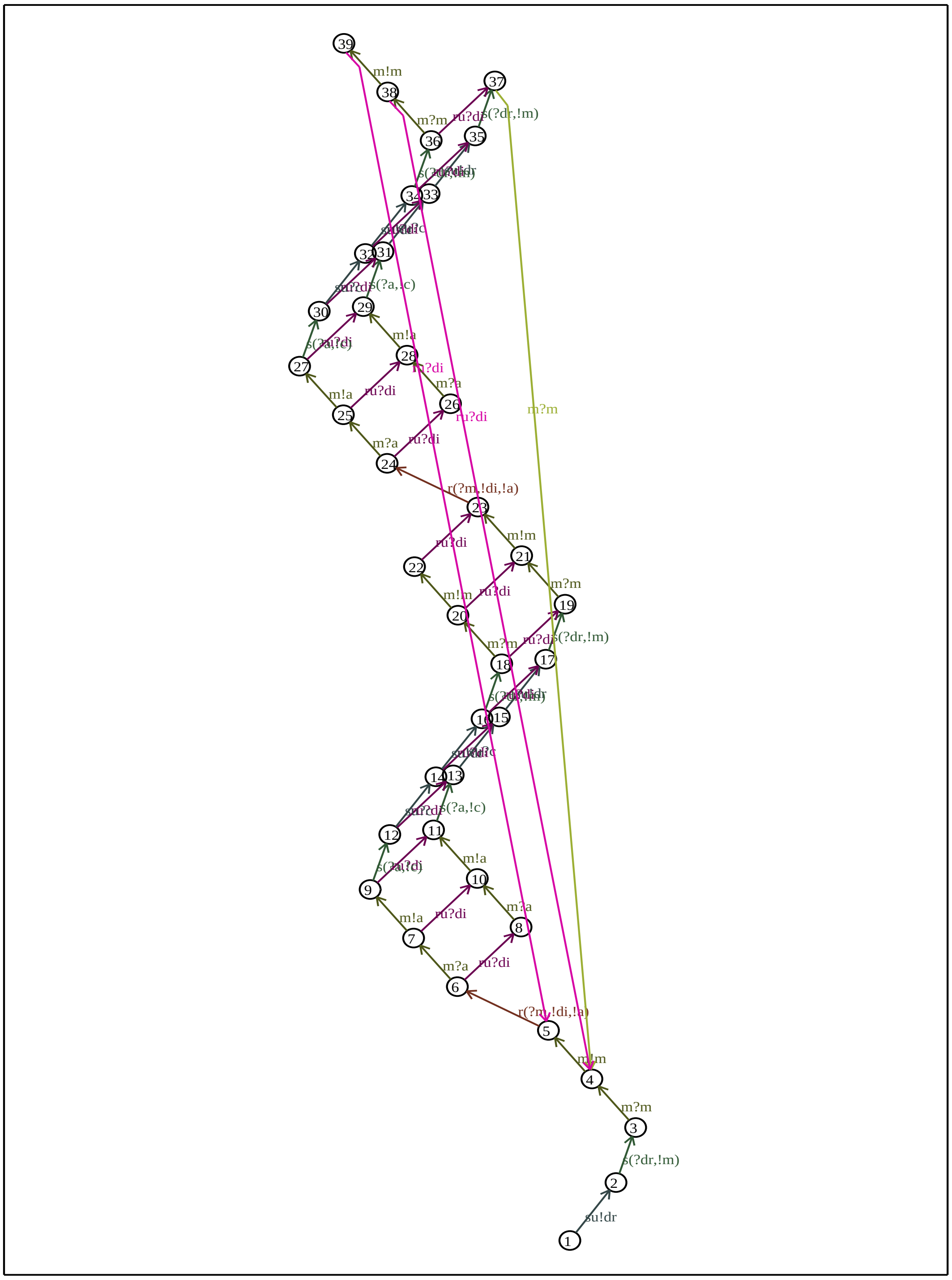

Figure 3: State graph of the alternating-bit protocol represented by Viscope

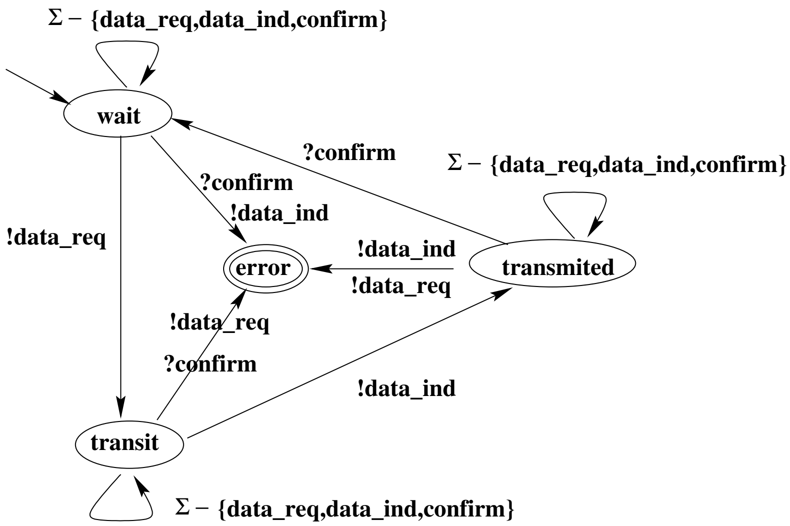

Figure 4: Automaton for the service property

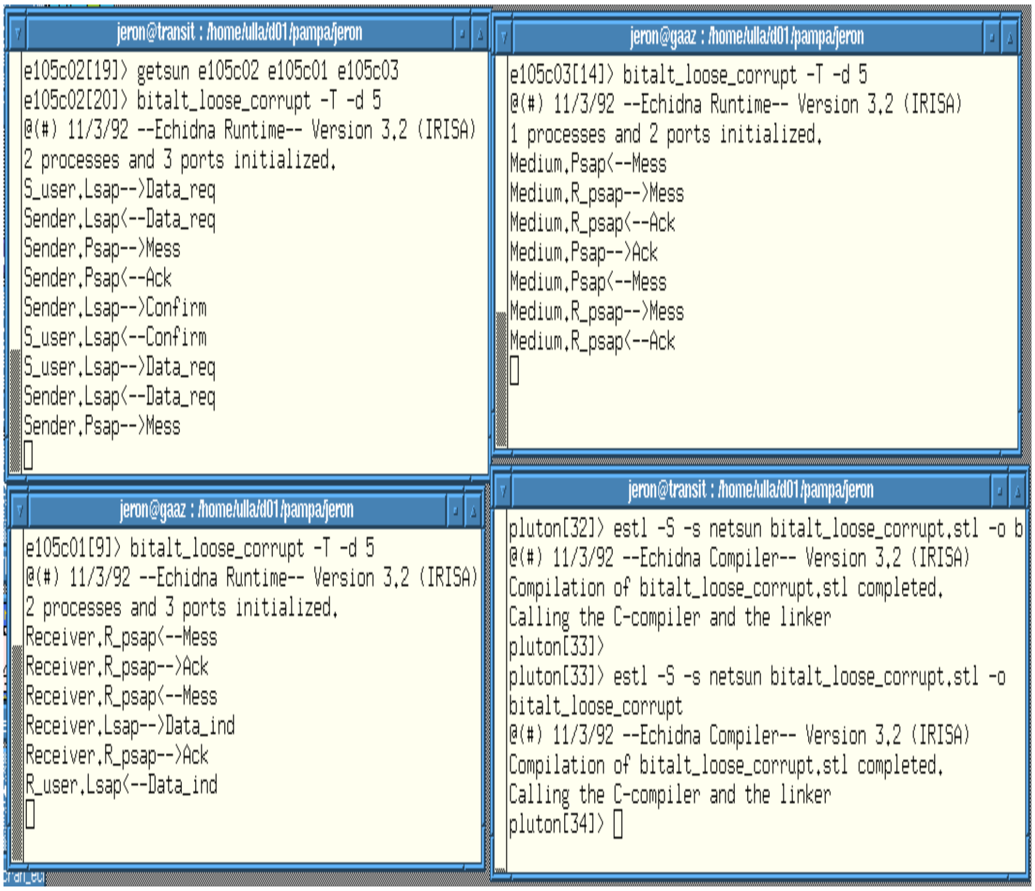

Figure 5: Distributed execution of the alternating-bit protocol on three machines

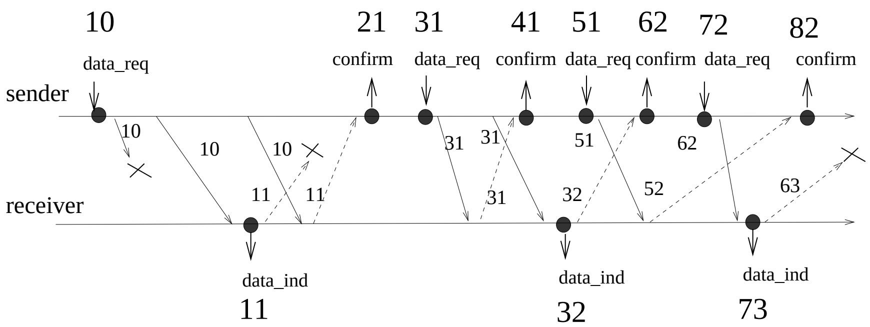

Figure 6: Time-stamps

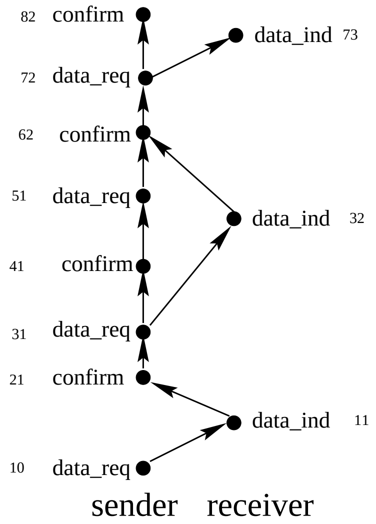

Figure 7: Hasse diagram

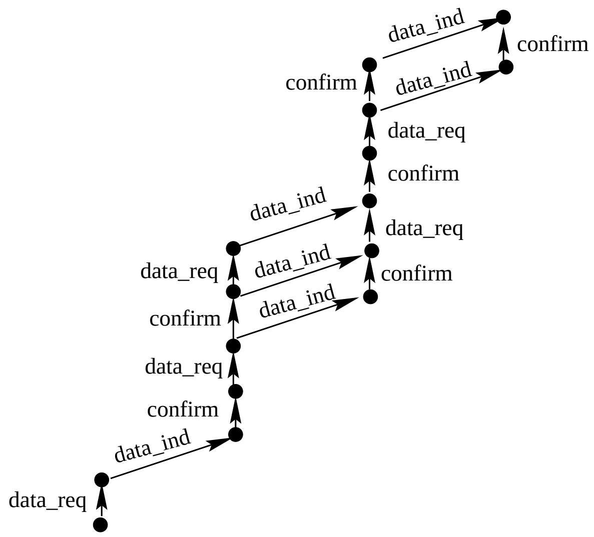

Figure 8: Lattice of all the possible observations