RZ 2996 (#93042) 03/09/1998
Computer Science/Mathematics 29 pages

Research Report

Towards A Framework for Handling Disputes in Payment Systems

N. Asokan, Els Van Herreweghen, Michael Steiner

IBM Research Division
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

IBM Research Division
Almaden · Austin · Beijing · Haifa · T.J. Watson · Tokyo · Zurich

Towards A Framework for Handling Disputes in Payment Systems

N. Asokan, Els Van Herreweghen, Michael Steiner

IBM Research Division, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

The ability to support disputes is an important, but often neglected aspect of payment systems. In dealing with dispute support, the first step is to define *how* to express various dispute claims. In this report, we present a language for expressing dispute claims in a unified manner, independent of any specific payment system. We illustrate how to support claims made in this system-independent language with evidence tokens from an example payment system. We describe an architecture for handling disputes, and present a design for incorporating dispute handling support into SEMPER [Wai96]. Our approach may be generalised to other services where accountability is a requirement.

1 Introduction

1.1 Importance of Disputes in Electronic Commerce

Services in an electronic commerce system involve more than one player. An effective system guarantees that if all players behave correctly ("honestly") according to some pre-defined protocol, each player obtains the services it expects. A system with *integrity* [PW96a] guarantees that, in addition, honest players are also protected against the incorrect behaviour of other players they do not trust. For example, in a payment system, an integrity requirement of an honest payer is that the payee receives at most the amount of value authorised by the payer.

Sometimes practical considerations may render it desirable to settle for a weaker form of integrity — the integrity requirement is modified as follows: even if the system is not able to prevent a dishonest player from causing "harmful" effects to the honest player(s), it must allow the honest player(s) to later prove the behaviour of the dishonest player. "Standing order" payments is an example. The payer instructs his bank to allow periodic value transfers requested by the payee (e.g., for paying utility bills). While the payer cannot prevent the payee from requesting more money than necessary (e.g., more than the amount of the monthly bill), he can prove the amount transferred by obtaining a statement from the bank.

Furthermore, electronic commerce transactions typically have legal significance in the real world. This means that even if a transaction is concluded successfully, there may be subsequent disputes about what happened during the transaction (or whether in fact an alleged transaction took place).

Thus, the ability of an honest player to win any dispute about a past or current transaction is often an important requirement.

1.2 Handling Disputes

Even those systems which have accountability as one of their major goals (signature systems such as RSA [RSA78], payment systems such as SET [MV97], or integrated electronic purchase systems such as NetBill [CTS95]), usually limit themselves to the generation and collection of evidence. Analysis of these systems may include proofs which demonstrate that the collected evidence is enough to win any disputes. It is assumed that such evidence can be used in some dispute resolution procedure external to the system. Obviously, this is not practical: it excludes the possibility of the system making its own decisions based on the outcome of disputes, or internally trying to recover from errors or failures (e.g., caused by loss of a network connection).

Moreover, evidence tokens are essentially part of the internal structure of the system; their structure and raw contents are not relevant outside the system. For instance, a payment receipt in the form of a digital signature is outwardly just a string of bits. The semantics to the evidence is added by the system itself. Outside the system (that is, from the point of view of the user of a system), what is necessary is to know what the evidence *means*, and how it can be *used* in disputes. Thus, a system providing a primary service (e.g., a payment service) should also support a *dispute service*. The dispute service specifies how to initiate and resolve disputes for the given primary service.

In a dispute, there is a set of (one or more) players called *initiators* who start the dispute and another set of players called *responders* who participate in it. A special player called the *verifier* or *arbiter* makes, or helps in the making of, the final decision regarding disputes, according to some well-defined procedures which can be verified by anyone. The initiator(s) try to convince the verifier of a *claim*. Initiators may support their claims by producing evidence or engaging in some sort of a proof protocol. Responders may attempt to disprove the claims. The verifier analyses the claims made and the evidence presented. This analysis may lead to a judgment as to whether a dispute claim is valid or not.

Completely automated dispute resolution may not always be feasible or even desirable. In fact, complete automation should not be our intent. Our dispute handling service should instead be used as a tool in human-driven dispute resolution. There are several projects that attempt to investigate the legal aspects of human-assisted online arbitration [Kat96]. The verifier in the dispute handling service can be used by an expert witness in court in order to support his testimony. Or, it can be used by an entity like the Online Ombuds Service [Kat96] which is not a legally competent authority but helps players resolve their disputes. In these cases, the verifier does not make a final decision. Instead, it presents an analysis of the evidence to a human judge. The verifier may even be one of the players themselves, trying to settle a dispute in a friendly way without going to court, or trying to convince itself of which disputes it can win. For example, if a customer claims to a merchant that he has already made a payment, the first step would be to try resolving the dispute without using an external verifier. In this case, the merchant may ask the customer to prove his claim: they run the dispute resolution protocol with the merchant playing the role of the verifier. Even before complaining to the merchant, the payer may have run the dispute resolution protocol, playing the role of the verifier himself, so as to confirm that his claim is provable. Note that a verifier's analysis of the evidence presented depends on its policy, trust assumptions, and context. Therefore the same evidence may be interpreted differently by different verifiers.

The first step in developing a coherent approach to dispute handling is to figure out how to define a dispute handling service given the description of some primary service. To keep the problem tractable, we focus on handling disputes in payment systems - we will however attempt to stay general as far as possible so that the approach outlined here has the potential of being applicable to other types of generic service definitions as well. In [APASW98], a generic payment service was developed, abstracting away details specific to individual payment systems. Such a generic service definition enables the users (either human users or applications acting on their behalf) to invoke payment services in a system-independent fashion. Clearly, the advantages of having a generic service definition for payments would be diminished if the handling of disputes were to be system-specific. Therefore, our aim is the development of a unified framework, rather than focussing on a specific payment system.

In Section 2, the problem of how to express dispute claims is studied. As we saw, dispute claims in a generic service must be generic as well. We will use the generic payment service developed in [APASW98] as an example to illustrate dispute claims. In Section 3, the problem of how to map evidence tokens to dispute claims is investigated. This mapping is payment system-specific and needs to be done internally in every payment system. As an example, we will use a simplified version of the iKP [BGH$^+$95] payment protocol.[1] An overall architecture for dispute handling, including a general dispute resolution protocol, is also described. In Section 4, the architecture is illustrated with a concrete design.

[1] SET is the proposed standard for credit-card payments on the Internet. However, we will use iKP here since its simplicity helps illustrate the approach more clearly.

2 Expressing Dispute Claims

2.1 Model and Notation

We use the ISO-OSI approach of modelling a distributed system [Lin83] by defining the **service** provided by it. The purpose of a system is to provide some capabilities to its users. In a distributed system, users are typically not co-located. A user accesses a system at a **service access point**. The interaction between a user and the system is carried out by invoking **service primitives** at access points. A service primitive can be either an input to the system or an output from the system. Each service primitive may be associated with a set of **service parameters**. The service parameters represent the information exchanged between the user and the system during a service primitive invocation. The **service** of the distributed system is essentially a statement of the capabilities provided by it. In concrete terms, the service is defined by the interface between the users and the system as the complete set of service primitives.

2.2 What to dispute?

Consider a payment system which implements the services defined by the generic payment service described in [APASW98]. Assume that the system has already been appropriately initialised. The primary purpose of the generic payment service is the transfer of value from payer to payee. The description of value transfer services is given in Table 1.

Primitive	Input	Output	Description
	Base Services		
pay	*payee, amount, options,ref.* [a]		send a payment
receivePayment	*[payer,] [amount,] options, ref.*	*payer, amount*	receive a payment
reversePayment	*transaction record*		ask/get a refund
reverseReceivedPayment	*transaction record*		make a refund
Additions for cash-like model			
withdraw	*amount, options, ref.*		load money into purse
deposit	*amount, options, ref.*		unload money from purse
Additions for cheque-like model			
receiveRawPayment	*[payer,] [amount,] options, ref.*	*payer, amount*	receive a payment (defer authorisation)
authorise			authorise a previous raw payment
capture	[*amount*]		capture a previous raw payment
multiCapture	*list of transaction records*		capture a set of previous raw payments

Table 1: Generic Payment Service: Value Transfer Services

[a] ref. allows to link this payment tightly to its context, e.g an on order and its description

To make a value transfer, the payer tells the system who the payee is, what amount is to be transferred, and certain other parameters.

- pay \$200 to BobAir ("#434: for flight 822 on Jan 19").

In the generic payment service, this is represented by the service primitive pay which takes the following pieces of information as parameters: payee, amount, ref (the external reference string enables the payment transaction to be linked to an external context). In order to complete the value transfer, the payee invokes a receive primitive with input parameter ref (optionally also payer and amount) and output parameters payer, amount. Figure 1 illustrates the interface events during a payment.

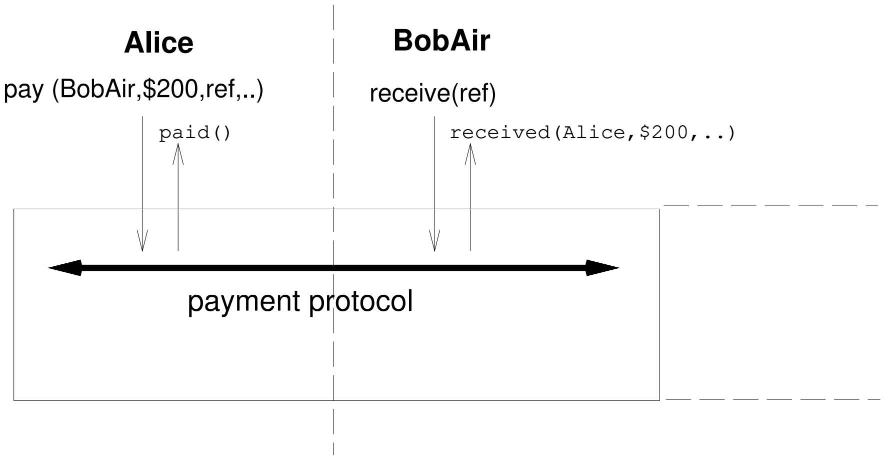

Figure 1: An Example Payment Transaction

Before formally defining a language to express dispute claims, let us attempt to get an idea of the kinds of claims that need to be expressed. What sorts of disputes, related to the above value transfer, does the payer (Alice) expect to be able to initiate and win? For example, Alice may want to claim that

- she paid \$200 to BobAir (perhaps because BobAir refused to send the tickets claiming no payment was made), or
- her payment was made before Jan 12 (perhaps because there was a deadline).

Some disputes may be about negative claims: for example, BobAir may want to prove that

- BobAir did **not** receive \$200 from Alice.

In other words, dispute claims are statements about the characteristics of value transfer. These characteristics are determined by the service primitives used, together with their parameters, and additional contextual information (such as the time of value transfer).

In addition to the payer and payee, a financial institution may be involved in creating a digital representation of money or converting it back to real value. Thus the value transfer may involve two or more sub-protocols involving different pairs of players. For example, in the cheque-like model, the payer sends a "form" (e.g., a cheque or a credit card slip) to the payee using a payment protocol and the payee may use a deposit or capture protocol to claim the real money. This leads to two other types of dispute claims.

- Suppose BobAir makes an offer to Alice for a cheap ticket if she made the payment before Jan 12. Alice goes through the steps of the payment protocol (e.g., sending a credit-card slip). However, BobAir changes his mind after receiving the credit-card slip — he does not "capture" Alice's payment. Alice cannot of course prove that the value transfer took place. But if she has a signed acknowledgement from BobAir, she can prove that the value transfer *could* have taken place without further help from Alice, if BobAir had wanted.

- Suppose Alice pays \$200 to BobAir using a debit card. Later, she finds an entry in her monthly statement indicating a debit of \$300. Alice may now want to start a dispute with the bank claiming that she approved a debit of only \$200. In other words, a single original transaction could lead to two different types of disputes: one involving the payer and the payee, and the other involving the payer and the bank.

2.3 Value Transfers as Primitive Transactions

Users expect a system to provide a certain service. Therefore, disputes in the system are about an instance of the service that was or could be provided. The primary service provided by the generic payment service is value transfer from one player to another. The model in [APASW98] assumes four types of players involved in value transfer: the payer, the payee, the issuer, and the acquirer. Value transfers between the issuer and acquirer are carried out over traditional banking systems; this transfer is outside the scope of the generic payment service. Thus, for the purpose of our disputes, we will consider the issuer and acquirer as a single entity, called the *bank*.

As shown above, we also need to express and conduct disputes about transfers between payer or payee and bank. This requirement leads us to follow the approach taken in [PW96b] of defining three different types of value transfer as shown in Figure 2.

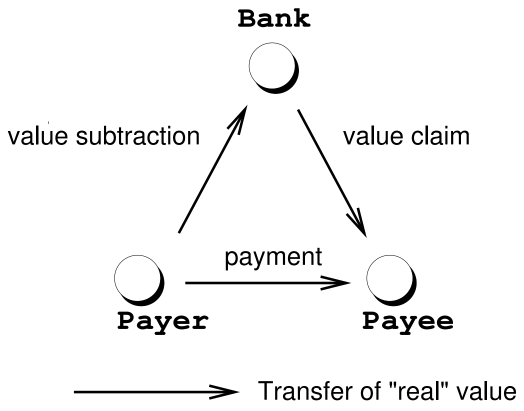

Figure 2: Value Transfer Transactions

- In *value subtraction*, a user allows the bank to remove "real value" from the user; this implies the user's right to spend "electronic value." In cash-like payment systems, this corresponds to the withdrawal of coins. In cheque-like payment systems, this is implicit in the payment itself. The players involved in a value subtraction are a bank and a user.

- In *value claim*, a user requests that the bank gives "real value" to the user. In cash-like payment systems, this corresponds to the depositing of coins. In cheque-like payment systems,

Primitive (See Table 1)	Types of Value Transfer
cheque-like model [a]	
pay	*payment, value subtraction*
receivePayment	*payment, value claim*
reversePayment	*payment, value subtraction*
reverseReceivedPayment	*payment, value claim*
receiveRawPayment	
capture, multiCapture	*value claim*
cash-like model	
pay	*payment*
receivePayment	*payment, value claim*
reversePayment	*payment*
reverseReceivedPayment	*payment, value claim*
withdraw	*value subtraction*
deposit	*value claim*

[a] authorise is not treated here since it does not involve a value transfer. But if the authorisation process involves non-repudiable evidence (e.g., guarantee from the bank to hold an amount for a certain time), it may be relevant in disputes. It would correspond to the *feasibility* of a value transfer.

Table 2: Types of Value Transfer

this is either part of the payment or corresponds to the separate capture transaction. The players involved in a value claim are a bank and a user.

- In *payment*, the payer transfers value to the payee. The players involved in a payment are the payer and the payee.

The relationship between the service primitives of Table 1, and the types of value transfer identified here is shown in Table 2.

We now define a *primitive transaction* as an instance of one of these value transfers. A primitive transaction represents a partial *view* of a subset of the players on the overall transaction.

In some cases, a primitive transaction actually corresponds to a protocol run of the underlying payment system. For example, a withdrawal protocol run in a cash-like system is a *value subtraction* primitive transaction (Figure 3). Thus, *value subtraction* completes independently of and before the actual payment. The *payment* and *value claim* complete after the actual cash payment protocol is run and the payee has deposited the coins with the bank.

In other cases, the primitive transaction is a purely *virtual* one and represents only the *view* of a subset of the players. For instance, a payment protocol run in a cheque-like payment system is seen by the set {payer, bank} as a *value subtraction* primitive transaction while it is seen by the set {payer, payee} as a *payment* primitive transaction (Figure 4).

Note that these subgroups correspond to the different applications that a player may use. In our example scenario, Alice may have used a standard on-line ticket purchase application. This application knows about BobAir, but may not know about Alice's bank (in fact, it need not even know which payment system is used to pay for the ticket). The purchase is viewed by this application as a *payment* primitive transaction, involving only Alice and BobAir. Alice may also have a banking application which she uses to interact with her bank in managing her bank account. This application does not know about BobAir. It will see the purchase as a debit from Alice's account — as a *value subtraction*.

Rather than referring to specific protocols or service primitives, we will state dispute claims in terms of whether or not the service defined by a primitive transaction did (or could) take place. If a

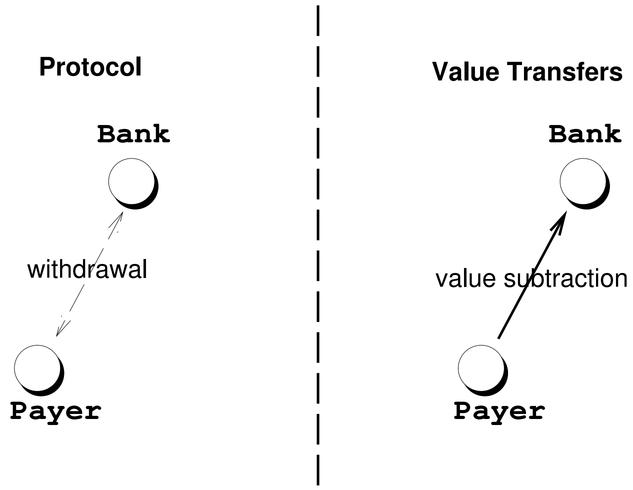

Figure 3: Primitive Transactions in a Cash-Like System for *withdrawal*

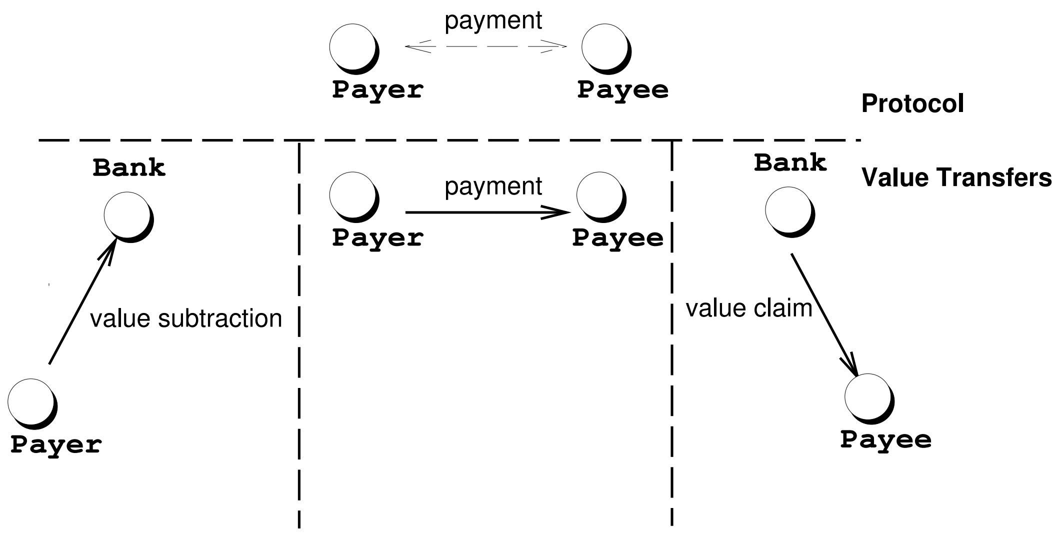

Figure 4: Primitive Transactions in a Cheque-Like System for *payment*

value transfer is reversed (e.g., the payee refunded the payer), it is equivalent to the value transfer not having taken place at all. However, it must still be possible to express a claim like "Alice did pay $200 to BobAir in the past" which must be true, even if the payment was later refunded by BobAir. If this is not sufficient, we can make reverse value transfers to be distinct primitive transactions.

2.4 Statements of Dispute Claims

2.4.1 Syntax

We express a statement of dispute claim as a formula in a first-order logic with certain modal extensions. The language of the logic consists of the following symbols: logical connectives, typed variables, typed constants, and relational connectives (which are functions of variables/constants of the appropriate type).

There are three types of variables: primitive transaction (pt), roles, and attributes. The pt variable can take its value from a well-defined enumerated set. In the case of the payment service, this set consists of PAYMENT, VALUE_SUBTRACTION, and VALUE_CLAIM. Each primitive transaction has a set of well-defined *role variables* associated with it. For example, PAYMENT has payer and payee as associated role variables. The role variables belong to a type called *id_val*, which represents distinguished names according to some well-defined naming scheme (e.g., account numbers or certified e-mail addresses). Each primitive transaction also has a well-defined set of *attribute variables* associated with it. For simplicity, we assume that all value transfer primitive transactions have the same set of attribute variables: amount, time,[2] and ref. The attribute variables are typed — they take their values from the appropriate domains. In the case of the payment service, we assume that amount, time, and ref take values from domains named *amount_val, time_val, and ref_val* respectively. Table 3 summarises the set of variables for the generic payment service. Each attribute,

Primitive Transaction	Roles	Attributes
VALUE_SUBTRACTION	user, bank	amount, time, ref
VALUE_CLAIM	user, bank	amount, time, ref
PAYMENT	payer, payee	amount, time, ref

Table 3: Dispute Claim Variables for the Generic Payment Service

depending on its type, has a finite set of relational operators associated with it. Table 4 lists the variables in the generic payment service, their domains, and applicable relational operators. We also allow two logical connectives: $\wedge$ (conjunction) and $\neg$(negation), a parenthetical operator for specifying precedence, and modal operators called **can_without**, **could_without**, **once**, **always** and **never**. A comma indicates concatenation.

The rules to construct valid dispute claim statements are described in the family of grammar specifications shown in Table 5. Angle brackets indicate place-holders which must be instantiated to derive a concrete grammar. Ellipses indicate that in a concrete instantiation, the rule may contain a repetition of same type of terms as the preceding term. Given a particular service, such as the

[2]There may be several different timestamps involved; for simplicity, we assume that there is only one instant at which the transaction is considered to have taken place.

variable	domain	relational operators
pt	{PAYMENT, VALUE_SUBTRACTION, VALUE_CLAIM}	$=$
$<role>$	*id_val*	$=$
amount	*amount_val*	$<\ \leq\ =\ \geq\ >$
time	*time_val*	$<\ \leq\ =\ \geq\ >$
ref	*ref_val*	$=$

Table 4: Attributes and Operators of Primitive Transactions

claim	::=	role **claims** claim_stmt
claim_stmt	::=	modal_stmt $\|$ $\neg$ modal_stmt $\|$ (modal_stmt) modal_stmt $\wedge$ modal_stmt
modal_stmt	::=	possibility_stmt $\|$ certainty_stmt $\|$ basic_stmt
certainty_stmt	::=	**always** basic_stmt $\|$ **never** basic_stmt
possibility_stmt	::=	role_set **could_without** role_set basic_stmt $\|$ role_set **can_without** role_set basic_stmt $\|$ **once** basic_stmt
role_set	::=	role $\|$ role, role_set
basic_stmt	::=	role_part $\wedge$ attr_part
role_part	::=	pt=$<$*primitive tx*$>$ $\wedge$ role_list $\|$ ...
role_list	::=	role=*id_val* $\|$ role=*id_val* $\wedge$ role_list
attr_part	::=	true$\|$ attr_val_pair $\wedge$ attr_part
attr_val_pair	::=	$<$*attr*$>$ $<$OP$>$ $<$value$>$ $\|$ ...
role	::=	$<$*role*$>$ $\|$ ...

Table 5: Family of Grammars for the Dispute Claim Language

generic payment service, the place-holders are instantiated, leading to a concrete, single grammar, as shown in Table 6. From now on, we will simply write PRIMITIVE_TRANSACTION_NAME to denote the predicate 'pt=PRIMITIVE_TRANSACTION_NAME.' Also, when a conjunction ($\wedge$) is obvious, we omit it. For example, 'PAYMENT payer =Alice payee =BobAir' is shorthand for 'pt=PAYMENT $\wedge$ payer =Alice $\wedge$ payee =BobAir'.

<table>
<tr><td>claim</td><td>::=</td><td>role claims claim_stmt</td></tr>
<tr><td>claim_stmt</td><td>::=</td><td>modal_stmt || ¬ modal_stmt || (modal_stmt)</td></tr>
<tr><td></td><td></td><td>modal_stmt ∧ modal_stmt</td></tr>
<tr><td>modal_stmt</td><td>::=</td><td>possibility_stmt || certainty_stmt || basic_stmt</td></tr>
<tr><td>certainty_stmt</td><td>::=</td><td>always basic_stmt || never basic_stmt</td></tr>
<tr><td>possibility_stmt</td><td>::=</td><td>role_set could_without role_set basic_stmt ||</td></tr>
<tr><td></td><td></td><td>role_set can_without role_set basic_stmt ||</td></tr>
<tr><td></td><td></td><td>once basic_stmt</td></tr>
<tr><td>role_set</td><td>::=</td><td>role || role, role_set</td></tr>
<tr><td>basic_stmt</td><td>::=</td><td>role_part ∧ attr_part</td></tr>
<tr><td>role_part</td><td>::=</td><td>pt=PAYMENT ∧ payer=id_val ∧ payee=id_val ||</td></tr>
<tr><td></td><td></td><td>pt=VALUE_CLAIM ∧ user=id_val ∧ bank=id_val ||</td></tr>
<tr><td></td><td></td><td>pt=VALUE_SUBTRACTION ∧ user=id_val ∧ bank=id_val</td></tr>
<tr><td>attr_part</td><td>::=</td><td>true|| attr_val_pair ∧ attr_part</td></tr>
<tr><td>attr_val_pair</td><td>::=</td><td>amount relop amount_val || time relop time_val ||</td></tr>
<tr><td></td><td></td><td>ref=ref_val</td></tr>
<tr><td>relop</td><td>::=</td><td>< || ≤ || = || ≠ || ≥ || ></td></tr>
<tr><td>role</td><td>::=</td><td>payer || payee || user || bank</td></tr>
</table>

Table 6: Grammar for the Payment Dispute Claim Language

2.4.2 Semantics

First, let us try to capture the intuitive semantics of our dispute claim language. During the execution of a protocol, the system as a whole goes through a series of well-defined *global states*. The global state consists of the initial secrets (e.g., private keys) of all the players involved, all message exchanges up to that point, and all sources of randomness. Therefore it has enough information to assign values to all the variables that can possibly appear in a dispute claim phrased in our claim language. Given a dispute claim phrased in our claim language, a verifier who knows the entire global state can decide with certainty if the claim is true or not. However, it is extremely unlikely that any verifier can know the entire global state (e.g., private keys of other players). The goal of dispute resolution is for the verifier to attempt to partially reconstruct the *sequence of global states* (along with as much of their contents as possible or necessary) the system has gone through, arriving at the *current state*. A verifier can do this using the *evidence* presented to it. Based on interpretation of the evidence (which is a payment system-specific function), the verifier can assign values to *some* of the variables. Such a *partial assignment* may be contingent on the trustworthiness of the entities involved in the creation of the evidence. The claim is evaluated with respect to the current state, and/or the sequence of states traversed so far.

Once a sufficiently complete interpretation of a state is available, the verifier can determine if a given basic_stmt s is true in that state. The meanings of the modal operators are intuitive. The statement "**always** s" is true at a state S if s is true in S as well as in every state reachable from S. The meaning of "**never** s" is analogous. The statement "P **can_without** Q s" is true in S if there is a state S' where s is true, *and* it is possible for P to cause the transition from S to S' without any action from Q. Given a path $p = \{S_0, \ldots S_n\}$, the statement "P **could_without** Q s" is true in p if the following two conditions are satisfied: (a) "P **can_without** Q s" is true at some state S in p, and (b) if, at some later state in p, it was no longer possible to reach a state where s is true, then P was responsible for this change.

Now, let us try to define the semantics more formally. Our model consists of a set of *states* S, a set of *roles* R, a set of *transitions* $T \subseteq S \times S$, and an interpretation L which assigns a value of the appropriate type to variables in the language.

A role uniquely identifies a service access point (e.g., payer). We assume that there is an infrastructure that enables a verifier to unambiguously identify and authenticate the identity of a player (which is some value from the domain *id_val*) playing any given role. We assume that a multi-party protocol can be described by a directed acyclic graph (DAG), the edges of which correspond to message transmissions between the players, and the nodes correspond to internal global states. We can capture this property by defining that each state has a cardinality, defined by the function $card() : S \rightarrow N$, where N is the set of natural numbers. The cardinality is used to impose a partial temporal order over S. If $(S_1, S_2) \in T$, then $card(S_2) > card(S_1)$. If the protocol is specified in the form of local finite state machines, they can first be unwound into a set of DAGs which can then be combined into a single DAG. The sender of a message is the agent associated with the edge that represents the message in the DAG. A function $agent() : T \rightarrow R$ identifies the role associated with a given transition. (Recall that an interpretation will associate an *id_val* with a role, thereby associating an *id_val* with each transition as well.) Given a *path* p, in the form of a sequence of states $\{S_0, S_1, \ldots, S_n\}$, the function $agents(p)$ returns the union of $agent(S_i, S_{i+1})$, for $i = 0 \ldots n-1$.

The verifier has a payment system-specific evaluation function which can be used to associate a partial assignment with a given state. The relational operators have the usual semantics. Thus, given a sufficiently complete partial assignment, and a proposition involving an attribute, a relational operator, and a value, it is possible to evaluate if the proposition is true. Given a global state S with a partial assignment, and a basic_stmt s of the form "role_part attr_part" (where role_part and attr_part are conjunctions of propositions as described in the previous section), s is true in S, if role_part and attr_part evaluate to true after the partial assignment is made. Since the assignment is partial, the verifier may not always be able to decide whether a claim is true or not, for example, if the evidence supplied is incomplete.

Figure 5 illustrates the semantics of the modal operators. The figure shows the DAG description of a protocol, including the states where a certain basic_stmt s is true.

1. If the statement '**always** s' is true in a state (e.g., S_{100}), then s (as well as '**always** s') is true in that state and in all states reachable from it, in all possible paths. Similarly, if '**never** s' is true in a state (e.g., S_3, S_{201}), then '**not** s' (as well as '**never** s') is true in that state and in all states reachable from it.

2. The statement 'P_2 **can_without** P_1 s' is true in S_1 because P_2 can cause the transfer to S_{100}.

3. The statement 'P_2 **could_without** P_1 s' is true in the path $\{S_0, S_1\}$ because of 2. It is also true in the path $\{S_0, S_1, S_{110}\}$ even though s itself cannot be true in S_{110} or any state reachable from S_{110}. This is because, in S_1 it was still possible to reach a state where s would have been true (S_{100}) and it was P_2 which chose not to effect that transition.

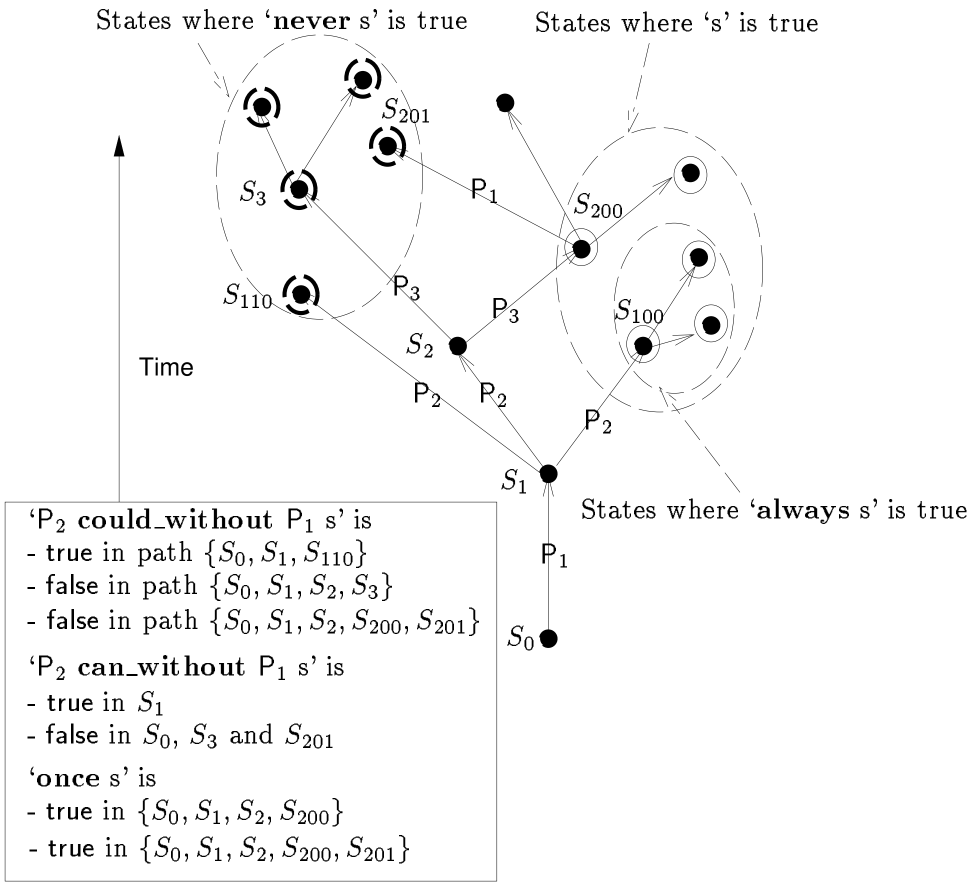

Figure 5: Semantics of Dispute Statements

4. Given a path, the statement '**once** s' has the usual meaning in linear temporal logic — for example, '**once** s' is true in $\{S_0, S_1, S_2, S_{200}\}$ and $\{S_0, S_1, S_2, S_{200}, S_{201}\}$.

We now define the semantics of modal operatores more formally. We first define a valid path as:

For a path $p = \{S_0, S_1, \ldots S_n\}$,
$\quad$ $valid\_path(p)$ iff
$\quad\quad$ $(S_i, S_{i+1}) \in T$, $i = 0 \ldots n-1$

In the following definitions, paths are implicitly assumed to be valid paths. The semantics of the **can_without** operator are now defined as follows:

$$
\begin{array}{l}
S \vdash PSET\ \textbf{can\_without}\ QSET\ s, \text{ iff } \exists\ S_n \text{ such that} \\
\qquad\qquad S_n \vdash s \\
\qquad \wedge \quad \exists\ p = \{S_0 = S, S_1, \ldots S_n\} \text{ such that} \\
\qquad\qquad agent(S, S_1) \in PSET \\
\qquad\qquad \wedge \quad QSET \cap agents(p) = \phi
\end{array}
$$

That is, given a state S, if there is a valid path p leading to a state S_n, such that s is true in S_n, and the first transition in p can be made by a member of $PSET$ and no one from $QSET$ is required to make any transition in p, then the statement '$PSET$ **can_without** $QSET$ s' is true in S. The **can_without** operator is used to make a statement about the possible future states of the system. In contrast, the **could_without** operator is more general. It is defined with respect to a path (more precisely, with respect to a state and a *specific* path leading to that state). It can be used to make a statement about the system at the end of the path about where it *can* go in the future, as well as where it *could have* gone in the past. The semantics of the **could_without** operator can be defined in terms of the **can_without** operator:

$$
\begin{array}{l}
\text{For a path } p = \{S_0, S_1, \ldots S_n = S\}, \\
\qquad p \vdash PSET\ \textbf{could\_without}\ QSET\ s, \text{ iff} \\
\qquad\qquad S \vdash PSET\ \textbf{can\_without}\ QSET\ s \\
\qquad \vee \quad \exists\ S_i \in p, i = 0 \ldots n-1 \text{ such that} \\
\qquad\qquad S_i \vdash s \\
\qquad\quad \wedge \quad S_{i+1} \vdash \neg\, s \\
\qquad\quad \wedge \quad agent(S_i, S_{i+1}) \in PSET \\
\qquad \vee \quad \exists\ S_i \in p, i = 0 \ldots n-1 \text{ such that} \\
\qquad\qquad S_i \vdash PSET\ \textbf{can\_without}\ QSET\ s \\
\qquad\quad \wedge \quad \exists\ S_j \in p,\ j = i \ldots n-1 \text{ such that } S_j \vdash ALL\ \textbf{can\_without}\ QSET\ s \\
\qquad\quad \wedge \quad S_{j+1} \vdash \neg\ ALL\ \textbf{can\_without}\ QSET\ s \\
\qquad\quad \wedge \quad agent(S_j, S_{j+1}) \in\ PSET
\end{array}
$$

The set ALL represents the set of all roles for the primitive transaction referred to in s. The rule identifies three disjunctive conditions to evaluate the truth of the statement 's_{could}=$PSET$

could_without *QSET s*' with respect to a path p. The first disjunction says s_{could} is true if 's_{can}=*PSET* **can_without** *QSET s*' is true in the last state of p. The second disjunction says that if s was true at some state in p, but not at the state immediately following it, *and* the transition was caused by someone in *PSET*, then s_{could} is true with respect to p. The third disjunction is a little more complicated. The intent is to capture the following case. Sometime in the past, it was possible to reach a state where s is true (i.e., there was a path, say p' from some state S_i in p, making s_{can} true at S_i). The agents of p' consist of some members of *PSET* but none from *QSET*. It may also consist of *other* entities, not in either of the above sets. The statement '($PSET \cup agents(p')$) **can_without** s' should therefore hold at every state in p subsequent to S_j. If it fails to hold after a certain transition, *and* that transition was caused by a member of *PSET*, then we can assert that s_{could} is true in p. In the interest of simplicity, we relax the definition a little by using the set of all roles (ALL) instead of ($PSET \cup agents(p')$). The last disjunction captures this property.

Similarly,

$p \vdash$ **once** s, iff $\exists\ S \in p$ such that $S \vdash s$
$S_0 \vdash$ **always** s, iff $\forall\ p = \{S_0, \ldots\},\ p\ \vdash\ \Box\ s$
$S_0 \vdash$ **never** s, iff $\forall\ p = \{S_0, \ldots\},\ p\ \vdash\ \Box\ \neg\ s$

The notation "$p\ \vdash\ \Box\ s$" has the usual meaning in linear temporal logic: in the sequence of states p, the formula s holds true in every state. In Section 3.3, we will look at an example payment system to see how we can build a global state transition diagram.

2.4.3 Extensions

The claim language described above constitutes the set of symbols and grammar rules necessary for *specifying* dispute claims in the generic payment service. The language constructs deal with the possibility of multiple time-lines. Therefore, it belongs to the family of branching temporal logics [Eme90]. It does not include all possible temporal logic constructs: for example, the **until** operator. We have only included the constructs that appeared to be necessary to express the type of claims described earlier. However, we have included constructs like **can_without** and **could_without**, which are not standard branching temporal logic constructs.

The language can be extended as needed. One generic extension is the inclusion of additional primitive transactions which do not represent value transfer but may be relevant in disputes. One example is the transaction started by the **authorise** primitive in cheque-like systems. Another example is a "statement of account" transaction where a bank issues a statement indicating the current balance on a user's account.

A second type of generic extensions can be used to derive the language to describe the messages in a generic dispute protocol between the verifier and the player. One extension is the addition of a function operator **witnessed**, as described in Section 3.2 below. Another extension is to allow a statement to refer to multiple instances of one or more primitive transactions by using functions, such as summation (e.g., statements like "the sum of the amounts involved in all VALUE_SUBTRACTION primitive transactions between Jan 1 and Jan 20 is less than 400").

Finally, individual payment system adapters may introduce additional primitive transactions, and role/attribute variables necessary to describe their protocols. Such additions will not be visible at the interface to the payment dispute service; but they can be useful internally.

2.4.4 Examples

The five dispute claims mentioned in Section 2.2 correspond to the following statements in our language:

- PAYMENT payer=Alice payee=BobAir amount=$200
- PAYMENT payer=Alice payee=BobAir amount=$200 time < Jan12
- ¬ PAYMENT payer=Alice payee=BobAir amount=$200
- payee **could_without** payer PAYMENT payer=Alice payee=BobAir amount=$200 time < Jan12
- ¬ VALUE_SUBTRACTION *user*=Alice *bank*=CarolBank amount=$300 ref ="#434: for flight 822 on Jan 19:payment to BobAir"

If players have evidence proving that a certain value transfer transaction has reached a guaranteed final state, they may choose to make stronger claims, using the **always** or **never** operators. For example, if Alice has a receipt for the payment made using a payment system which does not support refunds, she may claim "**always** PAYMENT payer=Alice payee=BobAir amount=200" instead.

3 Supporting Claims with Evidence

3.1 Architecture for Dispute Handling

3.1.1 Overview

Recall that there are three types of players in a dispute: the *initiator* who starts the dispute by making a claim, the *verifier* who co-ordinates the dispute handling and possibly makes, or helps make, the final decision about the validity of the claim, and a set of *responders* who may be asked by the verifier to participate in the process.

At the access point of each player, there is a "user" part (which may be the human user, or an application program acting on his behalf) and a "system" part (which is an implementation of the dispute service: e.g., an iKP implementation of the generic payment dispute service). The verifier's system has two parts: the *inference engine* receives a claim, and associated non-repudiation tokens, analyses them, and determines the conditions under which the claim is true, and the conditions under which it is false; the *policy engine* uses the result of the first part to make a final decision. The policy engine may be a human arbiter, external to the dispute handling service. It needs to use trust assumptions. In the simplest case, this may be in the form of a blacklist of untrusted principals. It is possible that the trust assumptions require a more elaborate representation (for example, as in PolicyMaker [BFL96]).

Figure 6 illustrates the various interactions during a dispute transaction. To begin a dispute, the initiator constructs a dispute claim. The claim is sent to the verifier. The verifier's inference part decides which players need to prove which statements in order to resolve this dispute claim. This may involve interaction with the systems of players, asking them what evidence they can produce. The verifier's system then informs the players' systems of the statements they are required to prove.

Each player's system then engages in a proof protocol with the verifier's system. Recall that during the dispute protocol, the verifier's goal is to determine both the current state of the payment system, and the sequence of states through which it has progressed, and that the validity of the claim should be evaluated with respect to this state and sequence. At the end of the protocol, the

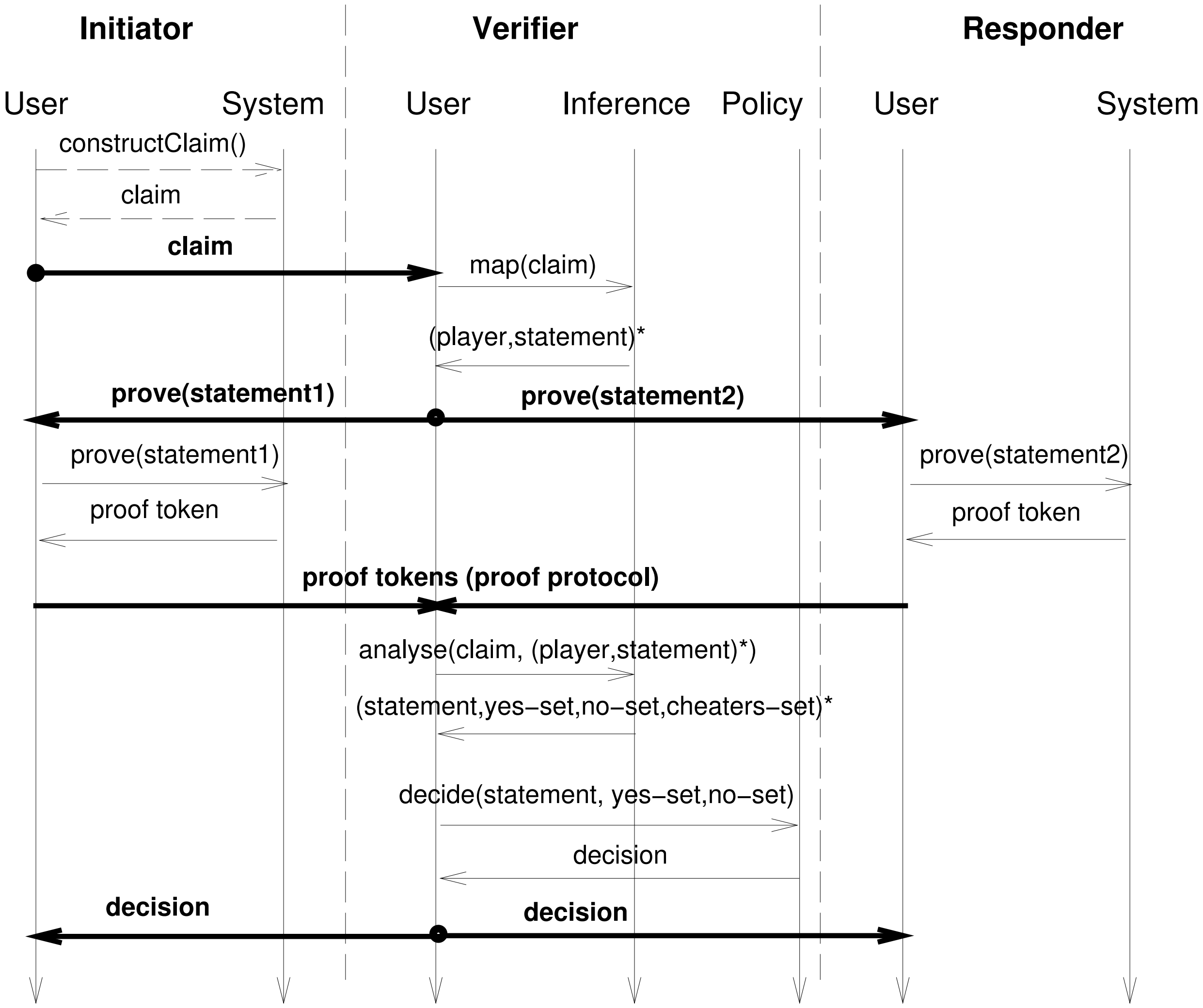

Figure 6: Basic Dispute Protocol

verifier's system returns an analysis of the claim. The analysis consists of a likely decision (yes or no) as well as the set of players who were witnesses to the decision and the set who were against it. If there is insufficient evidence, the verifier's system may throw an exception. The policy engine makes the final decision by combining this analysis with his trust assumptions.

This leads to the following requirements on the design:

- The verifier needs the following service primitives:
  - map: Takes a *claim* as input and returns a list of (*player*, *statement*) pairs. Each player is required to prove the corresponding statement.
  - analyse: Takes a *claim*, a set of players, and the statements they need to prove as input, engages in some proof protocol (maybe as simple as receiving one or more non-repudiation tokens, interpreting them, and making an inference on the statements proved), and returns the set of players according to which the claim is true, the set of players according to which it is false, and the set of players who have been found to be cheating.
  - decide: Takes a *claim*, and two sets of players (*yes*-set, *no*-set) as input, and returns one of yes, no, or cannot-decide, based on the verifier's trust relationships as output. This is the result of the dispute.
- Each player needs the following primitives:
  - constructClaim: which allows the user to construct a claim, and
  - prove: which takes a statement as input and attempts to prove it (maybe as simple as retrieving the pieces of evidence and returning them to the caller).

In Section 4 we present a design implementing this architecture.

3.1.2 Enhancements

We have limited ourselves to disputes in a generic payment service where there is only one service boundary. The system is "below" the boundary and the user or his application is "above" the boundary. Comprehensive electronic commerce frameworks such as SEMPER [Wai96] are structured into multiple layers. A payment usually takes place in the context of a higher layer transaction (e.g., a fair exchange) which in turn may take place in the context of a transaction in the layer above (e.g., an instance of an on-line purchase application). A dispute claim made in a higher layer needs to be suitably mapped to corresponding claims in the lower layers. The running of the dispute protocol needs to be co-ordinated among the different levels. This is left as an open problem.

3.2 Evidence and Trust

During a dispute, players have to support dispute claims by proving certain statements to the verifier. The ability to prove statements comes from pieces of evidence (evidence tokens) accumulated during a transaction of the primary service. The simplest form of evidence is a non-repudiation token that can be verified by anyone who has the necessary public keys, certificates, certificate revocation lists etc. The proof protocol in this case simply consists of presenting the token to the verifier. There can be more involved proof protocols, such as in "undeniable signature schemes" [CvA89]. In the following, we will assume only simple proof protocols consisting of the presentation of non-repudiation tokens.

Often, non-repudiation tokens cannot substantiate an absolute statement. In general, an evidence token corresponds to a non-repudiable assertion by one or more players that they believed the protocol reached a certain state (with an associated partial assignment). Such an assertion is a *witnessed statement*. During the dispute protocol, the verifier asks various players to prove witnessed statements (in the proof request messages in the dispute protocol of Figure 6). We define the language for these witnessed statements by building on our claim language in Section 2.4 and extending it with the following rule.

witnessed_stmt	::=	role **witnessed** asserted_stmt
asserted_stmt	::=	basic_stmt || certainty_stmt

The **witnessed** operator takes two parameters: an asserted_stmt s, and a role P. The statement "P **witnessed** s" is true if P has non-repudiably asserted that the transaction reached a certain state, with an associated partial assignment in which s evaluates to true. The ability to prove and verify such a non-repudiable assertion itself assumes a lower layer dispute service for non-repudiation.

The verifier's analyse method, after having collected the non-repudiation tokens proving the witnessed statements, returns a result of the form:

$\bigvee$ {[$\neg$] <claim_stmt> yes={...} no={...} }, cheating={...}

Each component of the disjunction contains either the original statement in the claim or its negation, and two optional sets of players: the set of players whose statements are in agreement with the conclusion and the set of players whose statements are contrary to the conclusion. The verifier will pick one of these disjunctions, depending on the set of players he trusts. If the evidence presented is insufficient to decide one way or the other, the system may raise an exception. Further, it may also be able to detect if some player has cheated (for example, if the same player has witnessed a certain statement and its exact opposite, it may imply that the player had cheated).

Note that a carefully designed, secure, payment system can be rendered insecure if the inference engine used by the verifier is wrong. It is important to make sure that the inference engine used not degrade the security of the protocol.

Consider as an example a dispute claim discussed earlier: Alice claims to have made a payment of $200 to BobAir. There is no way to say with absolute certainty whether the transaction actually took place. A receipt from BobAir proves the statement

BobAir **witnessed** PAYMENT payer=Alice payee=BobAir amount=$200

Whether a verifier can infer

PAYMENT payer=Alice payee=BobAir amount=$200

depends on the context, the trust assumptions of the verifier, and even on the actual payment system allegedly used for the value transfer. For example, in a dispute against BobAir, the verifier could accept a signed receipt from BobAir as sufficient evidence to conclude that the latter claim is true. However, if BobAir and Alice are in collusion and want to convince a third party (say the tax authorities) that a certain payment happened, BobAir's signed receipt alone is not sufficient. In this case, if the verifier trusts the bank, it can accept a signed statement from the bank as sufficient evidence. Thus an **analyse** result of the form:

PAYMENT <role_part> <attr_part> yes={bank} no={}

may allow the verifier to reach a final positive decision, whereas a result of the form:

PAYMENT <role_part> <attr_part> yes={payer,payee} no={}

cannot exclude collusion of payer and payee in trying to prove a payment.

A player may have gathered several evidence tokens during a particular transaction. Not all of them may be necessary to substantiate a dispute claim about some aspects of the transaction. Thus, given a statement to prove, the system should use only the minimal subset of the relevant evidence, in order to respect as much as possible privacy requirements of the different players.

Now, in Section 3.3, we will look at a simplified version of the iKP protocol to see (a) how statements can be supported with evidence and (b) how to derive system specific inference rules.

3.3 An Example: Evidence Tokens in iKP

iKP was designed as a solution for securing credit card payments over open networks. The original protocol was described in [BGH+95]. A more detailed definition with some improvements is available in [Tsu96]. In this section, we present a simplified version of the 3-party iKP (3KP) where all three players are assumed to have signature and encryption key pairs. We will see how the receipts gathered during an iKP protocol run can be mapped to dispute statements defined in Section 2.4 for the generic payment service.

3.3.1 Protocol Description

In our simplified version of iKP, there are three players: Customer (Payer), Merchant (Payee), and Acquirer (Bank). Before the transaction begins, the customer and merchant agree about the amount of payment ("price") and the description ("desc") of what the payment is for. The first half of Table 7 depicts the initial information of each player. To begin the transaction, the payee:

- generates two random nonces v and vc; these nonces will later be used as part of the receipt(s) from the payee,
- collects the common information ("com"). The common information consists of all the pieces of information that will be known to all the parties at the end of the transaction,[3] and
- generates a signature Sig_M containing hashes of the data items mentioned above and sends the signature along with any necessary information to the payer. This is the signed *offer* from the merchant.

The signature on the offer makes it non-repudiable. But this is not relevant to our discussion.

The payer then sends an *order* message. This is the authorisation by the customer to make the payment. The order is a signature Sig_C of the payer on two pieces of information,

- $\mathcal{H}(com)$, and
- an encryption of the price, the customer's account number ("CHI"), and $\mathcal{H}(com)$.[4]

[3] One item in the common information is a randomised hash of the description ($\mathcal{H}^R$(desc)). The payer and payee already know the description. The randomised hash allows the bank to confirm that the payer and payee agree on the description without the bank's having to know the actual text of the description

[4] Parts of the card-holder information is considered "secret" information (e.g., like a credit card number). The encryption in the *order* can be opened only by the bank — thus, the payee will not be able to determine CHI.

Data		Functions	
M	Merchant (payee)	$\mathcal{H}()$	A one-way hash function
C	Customer (payer)	$\mathcal{H}^R(m)$	A randomised one-way hash of m
A	Acquirer (bank)	$\mathcal{E}^R_X(m)$	A randomised encryption
TID	Transaction ID		of m with the public key of X
date	Time stamp	$S_X(m)$	Signature of X on m
price	Amount information		
desc	Description of payment		
v	Confirmation Authenticator		
vc	Cancellation Authenticator		
CHI	Card-holder Information		
com	Common information $(M, C, \text{price}, \text{TID}, \text{date}, \mathcal{H}^R(\text{desc}), \mathcal{H}(\text{v}))$		

Payment

$$\textit{offer}: C \xleftarrow{\text{TID}, \text{date}, \mathcal{H}(\text{v}), \text{Sig}_M = S_M(\mathcal{H}(com), \mathcal{H}(\text{v}), \mathcal{H}(\text{vc}))} M$$

$$\textit{order}: C \xrightarrow{\text{Enc} = \mathcal{E}^R_A(\text{price}, \mathcal{H}(com), \text{CHI}), \text{Sig}_C = S_C(\text{Enc}, \mathcal{H}(com))} M$$

$$\textit{auth-request}: M \xrightarrow{\text{TID}, \text{date}, \mathcal{H}(\text{v}), \mathcal{H}(com), \mathcal{H}^R(\text{desc}), \text{Enc}, \text{Sig}_M, \text{Sig}_C} A$$

$$\textit{auth-response}: M \xleftarrow{\text{yes}\|\text{no}, \text{Sig}_A = S_A(\text{yes}\|\text{no}, \mathcal{H}(com))} A$$

$$\textit{confirm}: C \xleftarrow{(\text{v}, \text{yes})\|(\text{vc}, \text{no}), \text{Sig}_A} M$$

Cancellation

$$\textit{cancel}: C \xleftarrow{\text{vc}} M$$

Refund

$$\textit{refund-request}: M \xrightarrow{\text{Sig}^{refund}_M = S_M(\mathcal{H}(com), \text{price}, \text{"refund"})} A$$

$$\textit{refund-response}: M \xleftarrow{\text{Sig}^{refund}_A = S_A(\mathcal{H}(com), \text{price}, \text{"refund"})} A$$

Figure 7: Simplified iKP Protocol

The payee forwards the *order* along with the *offer* to the acquirer requesting authorisation. The acquirer replies indicating whether the authorisation succeeded or not. For simplicity, let us assume that the acquirer immediately transfers the money from payer to payee if the authorisation is successful. The authorisation response Sig_A from the acquirer is signed.

Once the payee receives the authorisation response, he will send a confirmation to the payer. The confirmation contains the acquirer's authorisation response and the authenticator v. The pair (Sig_M, v) constitutes a receipt by the payee that he received the payment. If a payee decides to cancel a payment, he can issue a cancellation receipt to the payer by sending him vc. If the payer possesses the pair (Sig_M,vc), it is proof that the payee agreed to cancel the payment. If the payee cancels an already authorised payment, he can contact the acquirer and arrange for a refund.

The second half of Table 7 lists the pieces of information that are collected by the players at the end of a successful protocol run. Again, items within square parentheses are available only under certain circumstances.

Note that instead of using v and vc, the *confirm* and *cancel* flows from the payee to payer can be signed by the payee. The use of v and vc avoids the payee having to make two or more signatures by allowing the original signature to be "extended."

In the actual iKP protocol, the acquirer transfers the money from the customer to merchant during a "capture" transaction. The merchant can also capture a different (lower) amount than was previously authorised. The refund transaction is essentially a negative capture. Depending on the policies of the players, some of the receipts may be omitted. All these variations are not relevant to our discussion. Therefore they are left out from our simplified version.

Initial Information	
payer	desc, price, CHI
payee	desc, price, TID, date, v, vc
bank	
Collected Information	
payer	com, $\mathcal{H}$(v), Sig_M, [Sig_A], v‖vc
payee	com, Enc, Sig_C, [Sig_A], [Sig_A^{refund}]
bank	[com, Enc, CHI, Sig_M, [Sig_M^{refund}], Sig_C]

Table 7: Information of Players in a Completed iKP Transaction

3.3.2 Mapping iKP receipts to Dispute Statements

Table 7 lists the pieces of information known to each player at the end of a successful transaction. We can now try to extract evidence tokens from these pieces and identify the dispute claims they can support.

The payer can prove that a payment transaction took place by producing (Sig_M, v). However, the payment may have been cancelled later. The payee does not get a receipt from the payer acknowledging a cancelled payment. However, the cancellation must have involved the running of a refund protocol with the bank. Thus, an honest payee can produce (Sig_A^{refund}) to counter the payer's claim. Line 1 of Table 8 expresses this scenario. The rest of Table 8 is an exhaustive list of all the pieces of evidence in iKP and their mapping to dispute statements.

Role	Evidence	Statement		Possible Counter
		witness	Primitive Transaction	
payer	1. Sig_M, v, com	payee	PAYMENT	8
	2. Sig_A, yes, com	bank	VALUE_SUBTRACTION	10
	3. Sig_A, no, com	bank	**never** VALUE_SUBTRACTION	
	4. Sig_M, vc, com	payee	**never** PAYMENT	11
payee	5. Sig_C, com, Enc	payer	PAYMENT	3, 4, 10
	6. Sig_A, yes, com	bank	VALUE_CLAIM	10
	7. Sig_A, no, com	bank	**never** VALUE_CLAIM	
	8. Sig_A^{refund}, com	bank	**never** VALUE_CLAIM	
bank	9. Sig_M, Sig_C, com	payee	VALUE_CLAIM	8
	10. Sig_M^{refund}, com	payee	**never** VALUE_CLAIM	
	11. Sig_C, com, CHI	payer	VALUE_SUBTRACTION	3, 4

Table 8: Mapping Evidence to Dispute Statements in iKP

Notice that all parties have to reveal *all* the pieces of the common information ("com") to the verifier. It is possible that a dispute claim may involve only the "price" attribute. Nevertheless, the players are forced to reveal the "date" attribute as well. The description of what the payment is for ("desc"), however need not be revealed because "com" contains a commitment (in the form of a randomised hash of "com") of "desc" only. If the dispute was about the description as well, the commitment can be opened (by revealing the random factor used in computing the hash). If the same technique was applied to the rest of the common information, i.e., if it consisted of:

$$\mathcal{H}^R(M), \mathcal{H}^R(C), \mathcal{H}^R(\text{price}), \text{TID}, \mathcal{H}^R(\text{date}), \mathcal{H}^R(\text{desc}), \mathcal{H}(\text{v}),$$

it would have been possible to reveal only the necessary information to the verifier. In general, we recommend supporting the possibility of *least disclosure of information* as a principle of good practice in designing disputable protocols.

With the information in Table 8 and Figure 7, we can represent the global states in a run of the iKP payment protocol in the form of a DAG as in Figure 8.

In iKP, all three primitive transactions are completed at the same time. The states where the primitive transactions are succesfully completed are marked with a circle. The states where '**never** s' is true (s is any basic_stmt with any of the primitive transactions) are marked with a thick broken circle. Notice how the verifier can use this graph to implement the analyse method (Section 3.1): while 'PAYMENT *rest_of_the_claim*' is false in S_{300}, S_{400}, and S_7, the statement 'payer **could_without** payee PAYMENT *rest_of_the_claim*' is true in S_{300} and S_7 (but not in S_{400}).

4 A Concrete Design

4.1 Classes

In the generic payment service described in [APASW98], a payment system is incorporated into the generic payment service by means of an adapter. Each instance of the payment system available to a user is represented by a Purse in the generic payment service. Each adapter is required to extend the Purse class.

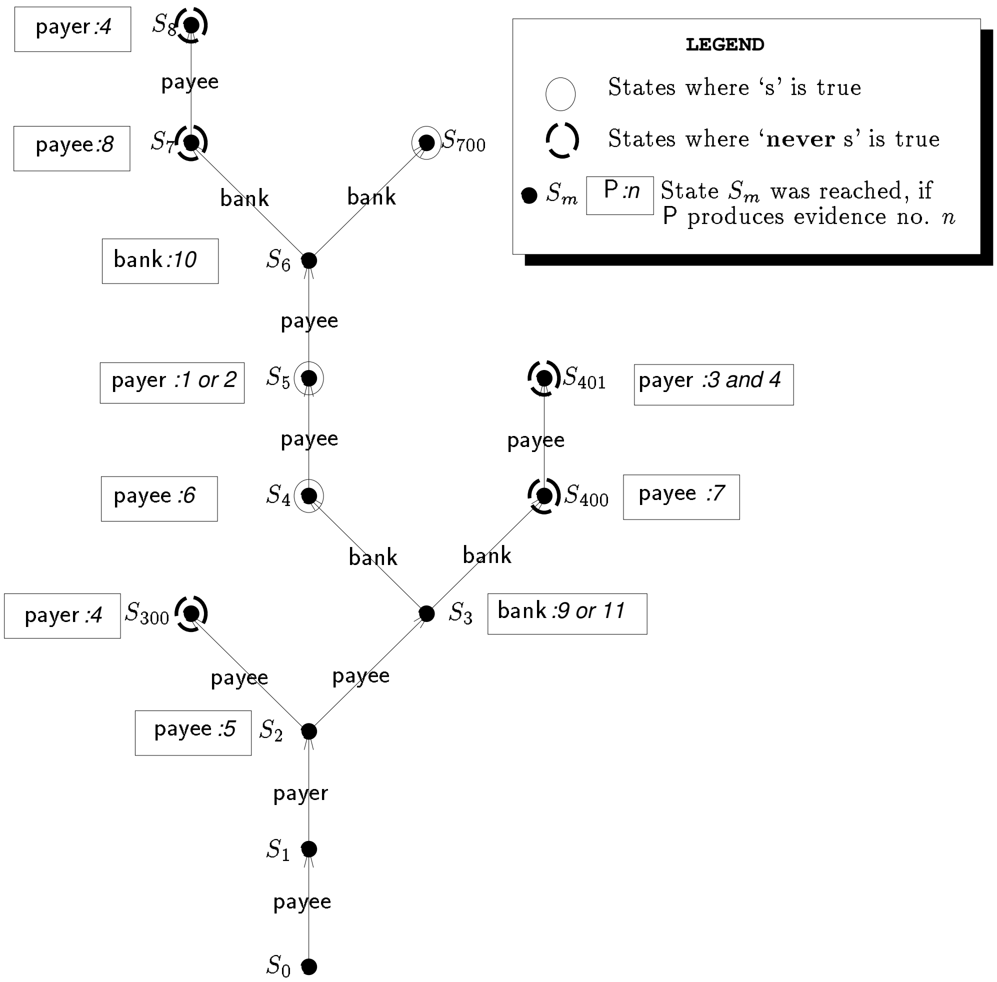

Figure 8: Global States in iKP

We begin by defining an abstract method startDisputeTx in the Purse class. This method will create a new dispute transaction, represented by the DisputeTransaction class. The DisputeTransaction hierarchy is shown in Figure 9. It provides methods to construct a claim, to start a dispute transaction, and since we adopt a token-based approach (described in [APASW98]) for running the protocol, a method to process protocol messages. Each adapter will be required to extend the DisputeTransaction class. The user's application will first identify the purse, ask it to create a dispute transaction, and then invoke the constructClaim on the dispute transaction. This method is responsible for constructing the claim. It could obtain the information it needs as input parameters or by interacting with the user. The constructClaim method also identifies a payment transaction record associated with the dispute. If one does not already exist (e.g., when an honest user wants to deny that she had made a payment), a new transaction record is created.

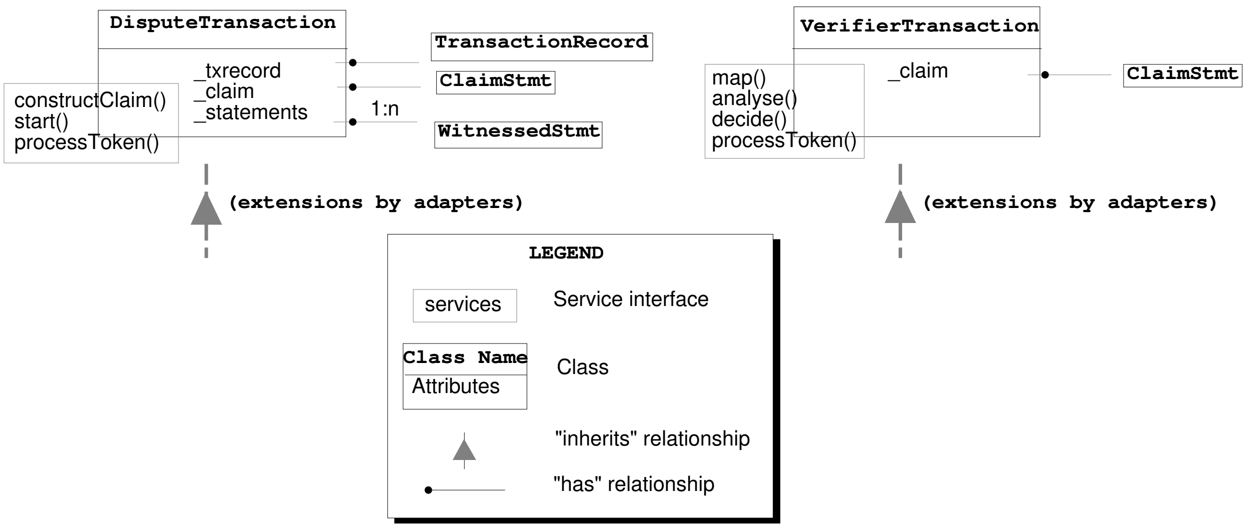

Figure 9: Class Hierarchy for Dispute Transactions

The class hierarchy for dispute statements is shown in Figure 10. A claim is an instance of the ClaimStmt class. According to the grammar, a claim_stmt is a boolean combination of instances of modal_stmt. In this design, we will only consider simple instances of modal_stmt, and negations. A modal_stmt is represented by the ModalStmt class which contains a basic_stmt, represented by BasicStmt. ModalStmt can be extended to PossibilityStmt or CertaintyStmt. A possibility_stmt or certainty_stmt has one of the modal operators enabled.

Individual adapters may extend the statements corresponding to the various primitive transactions (e.g., PaymentStmt); but they are not required to do so. New primitive transactions can be incorporated by introducing additional subclasses of the PaymentServiceStmt class. It is expected that disputes in other services will be handled by extending the BasicStmt class.

A statement consists of a role part and an attribute part. These are made of objects of the Role and AttrVal class hierarchies respectively. Figures 11 and 12 describe these class hierarchies.

The start method in DisputeTransaction will send the constructed claim to the verifier. The verifier can now instantiate a new VerifierTransaction. The map, analyse, and decide methods provide the functionality described in the previous section.

The thick solid grey arrows in the class hierarchy diagrams show the directions along which

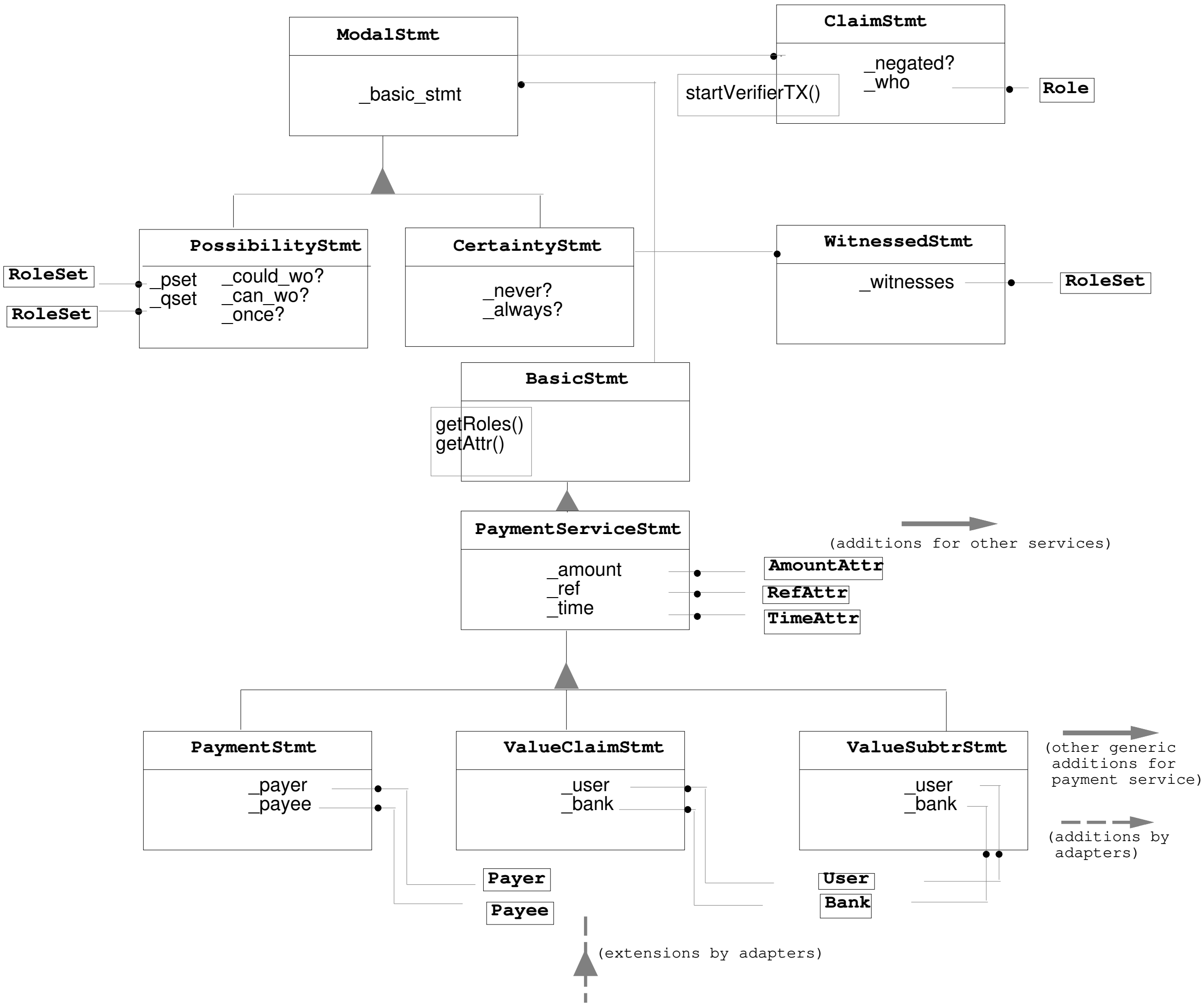

Figure 10: Class Hierarchy for Dispute Statements

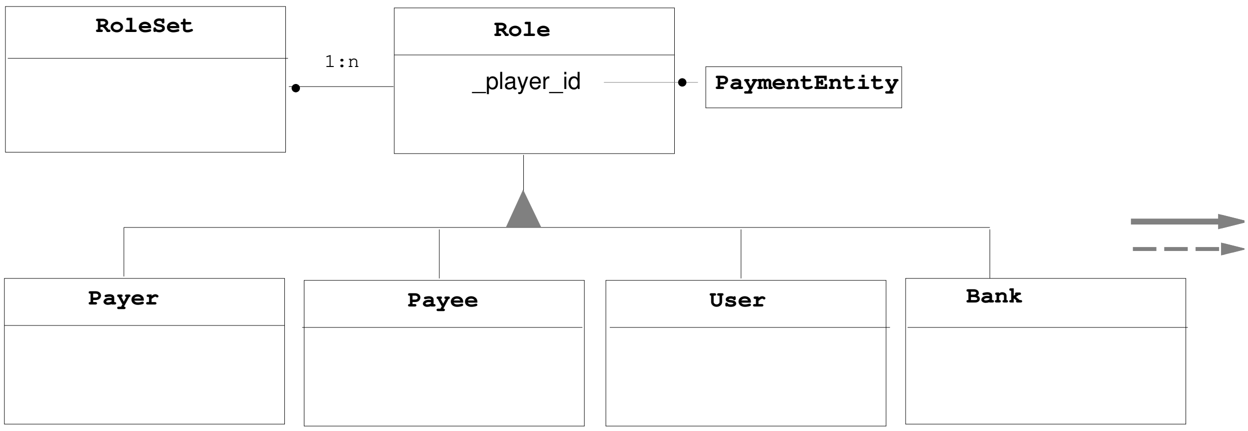

Figure 11: Class Hierarchy for Roles

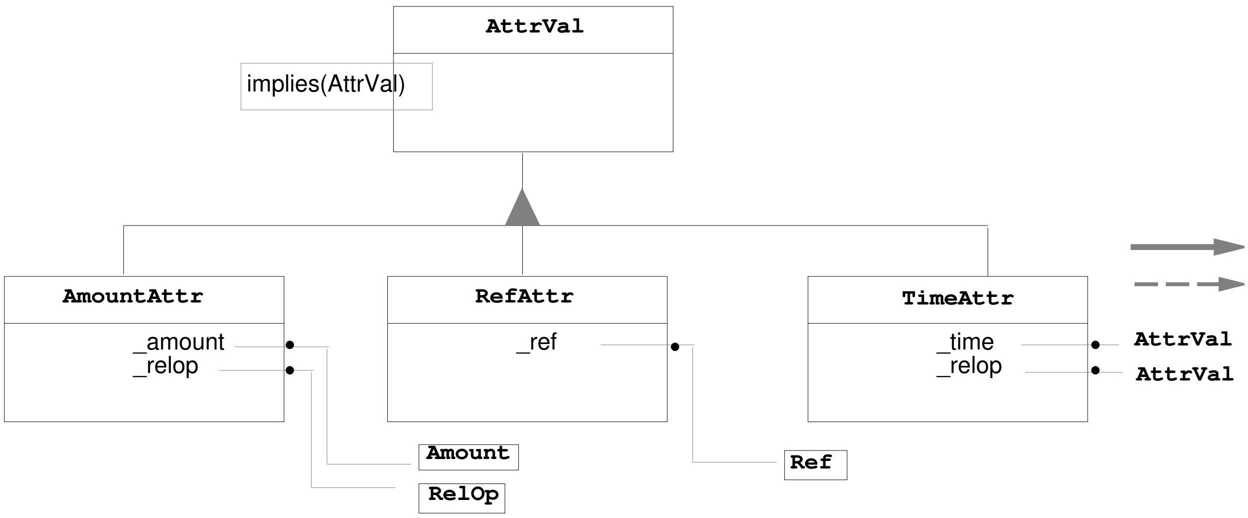

Figure 12: Class Hierarchy for Attribute-Value Pairs

generic extensions (such as adding a new primitive transaction) or extensions for other services (such as a generic signature service) can be made to the design.

4.2 Payment System Adapter Design and Implementation

4.2.1 Design

To support the dispute handling service in a payment system adapter, its designer should first identify the pieces of evidence collected during value transfer transactions and determine the statements they correspond to (as in Table 8).

4.2.2 Implementation

The thick broken grey arrows in the class hierarchy diagrams show the directions along which the adapters can extend the design. The implementation of dispute handling in an adapter consists of the following steps.

- Provide implementations of startDisputeTx methods in its Purse subclass,
- Extend the DisputeTransaction class; the startDisputeTx method should create and return an object of this subclass. This class should provide implementations for the abstract methods in the DisputeTransaction class. In particular, the implementations of the methods start and processToken should embody the mapping of evidence tokens to statements for this payment system, and
- Extend the VerifierTransaction class; the implementation of the map and analyse methods of this class should embody the default inference rules for this payment system.

5 Summary and Conclusion

We have shown why a generic dispute service is needed for payment systems. We developed a language to express dispute claims and applied it to an example payment system. Finally, we described a generic dispute handling protocol in the context of an architecture for dispute handling. The architecture can be realized using the design presented. A crucial part of the dispute handling framework is the verifier's inference engine. When a new payment system is adapted to the framework, the critical step is to identify the inference rules applicable to that system. In general, the process of deriving the inference rules is equivalent to proving the payment system correct. It is also necessary to verify that default inference rules do not violate the security of the system. At the current state of knowledge, this appears to be a hard task. Ideally, derivation of inference rules should be an integral part of the design process of the payment system.

On the other hand, the aim is not complete automation of dispute resolution. Thus, even with an incomplete set of inference rules, a payment system can be incorporated into the framework. The most trivial inference rule is “ask the human user!”

References

[APASW98] J. L. Abad-Peiro, N. Asokan, Michael Steiner, and Michael Waidner. Designing a generic payment service. *IBM Systems Journal*, 37(1):72–88, January 1998.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, Oakland, CA, May 1996. IEEE Computer Society,Technical Committee on Security and Privacy, The Institute of Electrical and Electronics Engineers IEEE Inc.: Computer Society Press.

[BGH+95] Mihir Bellare, Juan Garay, Ralf Hauser, Amir Herzberg, Hugo Krawczyk, Michael Steiner, Gene Tsudik, and Michael Waidner. iKP – a family of secure electronic payment protocols. In *First USENIX Workshop on Electronic Commerce* [USE95], pages 89–106.

[CTS95] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. NetBill security and transaction protocol. In *First USENIX Workshop on Electronic Commerce* [USE95].

[CvA89] David Chaum and Hans van Antwerpen. Undeniable signatures. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89*, number 435 in Lecture Notes in Computer Science, Santa Barbara, CA, USA, August 1989. Springer-Verlag, Berlin Germany.

[Eme90] E. A. Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 16, pages 995–1072. Elsevier Science Publishers B.V., 1990.

[Kat96] M. Ethan Katsch. Dispute resolution in cyberspace. In *Connecticut Law Review Symposium: Legal Regulation of the Internet*, number 953 in 28, 1996. Available from http://www.umass.edu/legal/articles/uconn.html.

[Lin83] Peter F. Linington. Fundamentals of the layer service definitions and protocol specifications. *Proceedings of the IEEE*, 71(12):1341–1345, December 1983.

[MV97] Mastercard and Visa. *SET Secure Electronic Transactions Protocol*, version 1.0 edition, May 1997. Book One: Business Specifications, Book Two: Technical Specification, Book Three: Formal Protocol Definition. Available from http://www.mastercard.com/set/#down.

[PW96a] B. Pfitzmann and M. Waidner. Properties of payment systems - general definition sketch and classification. Research Report RZ 2823 (#90126), IBM Research, May 1996. Submitted for Publication.

[PW96b] Birgit Pfitzmann and Michael Waidner. Integrity properties of payment systems, December 1996. Private Communication of work in progress; contact the authors for status of the work.

[RSA78] Ron L. Rivest, A. Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Journal of the ACM*, 21(2):120–126, February 1978. US Patent 4,405,829: Cryptographic Communications System and Method, Public Key Partners PKP.

[Tsu96] Gene Tsudik. Zürich iKP prototype: Protocol specification document. Technical report, IBM Research, February 1996. IBM Research Report RZ-2792.

[USE95] USENIX. *First USENIX Workshop on Electronic Commerce*, New York, July 1995.

[Wai96] Michael Waidner. Development of a secure electronic marketplace for Europe. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proceedings of the Fourth European Symposium on Research in Computer Security (ESORICS)*, number 1146 in Lecture Notes in Computer Science, Rome, Italy, September 1996. Springer-Verlag, Berlin Germany. also published in: EDI Forum 9/2 (1996) 98-106.