Analysing Time Dependent Security Properties in CSP using PVS

Neil Evans and Steve Schneider
Department of Computer Science
Royal Holloway, University of London

March 17, 2000

Keywords: Authentication Protocol Verification, Automated Theorem Proving, Timed Behaviour, CSP, PVS.

Abstract

This paper details an approach to verifying time dependent authentication properties of security protocols. We discuss the introduction of time into the Communicating Sequential Processes (CSP) protocol verification framework of [10]. The embedding of CSP in the theorem prover PVS (Prototype Verification System) is extended to incorporate event-based time, retaining the use of the existing rank function approach to verify such properties. An example analysis is demonstrated using the Wide-Mouthed Frog protocol.

1 Introduction

There are many methods that model and analyse security policies of distributed systems. Typically, the policies concerning communication are achieved using security protocols in which the agents of a system are trusted to provide a degree of secure communication across the system's network.

The complexity of security protocols and the size of distributed systems have often been too great for analyses without a great deal of abstraction. This can lead to over-simplification and the possibility of missing some size-dependent flaws. This is prevented in analyses in which a (potentially infinite) number of agents can engage in arbitrarily many (possibly concurrent) runs of a security protocol.

Model checking, via highly automated tools, has proved to be an invaluable ally to those who wish to analyse protocols by searching for attacks. However, justifying the correctness of a protocol from a model checking analysis is more difficult because it is usually impossible the explore the entire state space. Indeed, to explore any reasonable amount of state space, one is often required to approximate the model by considering individual runs of the protocol with the minimal number of agents.

Theorem proving, on the other hand, does not suffer from the state space problem. It is easier to justify protocol correctness, via a successful proof, than it is to extract an attack from an unsuccessful proof. Assuming that the protocols

of the future will not fail as readily as current protocols, these are obviously desirable qualities. The main disadvantage of theorem proving is the need for intense user intervention. Current tool support technology for theorem proving provides relatively little automation, though the situation is improving. Two general purpose theorem provers that have been used in the context of security protocol analysis are Isabelle [9], and PVS [8]; in this paper we will use the latter.

In these analyses, time is often abstracted from the protocol descriptions and from the properties that are proven about them. However, there are situations in which sensitivity to time is required for an appropriate analysis, either because the protocol exhibits some time-dependent behaviour (for example comparing a received value of a timestamp with the current time, or using timeouts and delays in its flow of control) or because the required property is concerned with time (for example, that a ticket accepted by an authorising agent has not yet expired).

Introducing time into a protocol analysis framework brings complications. The use of explicit time increases the state space of the system by a significant factor, which means that for model-checking either even simpler versions of the protocol need to be analysed, or else more powerful computers or larger time-scales are required to perform an appropriate analysis. The introduction of explicit time in theorem-proving approaches generally introduces time as an additional data-type, but the special nature of time imposes restrictions on its use, and requires that care should be taken to ensure that it is modelled in sensible and realistic ways. This is feasible when it is used in simple ways (such as simply providing timestamps and checking if they are recent) but the more complex the time-dependent behaviour, the less confidence we can have in ad-hoc approaches to introducing time.

In this paper, we shall introduce event-based time into the process modelling language CSP, and we shall extend the corresponding embedding of CSP in the theorem prover PVS [4] to incorporate this approach to time. One benefit of the CSP approach is that the theory of *timewise refinement* [11] allows results to be translated between the untimed and timed models, enabling verifications to be carried out at their most appropriate level of abstraction and then combined if necessary from different models. The timed model will be illustrated by proving a time dependent property of the Wide-Mouthed Frog protocol [2].

2 Review of Previous Work

2.1 Communicating Sequential Processes

CSP is a modelling language that allows the description of systems of interacting processes by means of a few language primitives. Processes execute and interact by means of performing *events* drawn from a universal set Σ. Some events are of the form $c.v$, where c represents a channel and v represents a value being passed along that channel. This allows messages to be communicated between processes.

The process *Stop* is a stopped process—it can perform no events. The process RUN_A is the process that will repeatedly be able to perform any event in the set of events A. The process $a \rightarrow P$ is initially willing to perform the event

a, and then behave subsequently as P. The input process $c?x \rightarrow P(x)$ will accept a value x along channel P and then behave subsequently as $P(x)$; The output process $c!v \rightarrow P$ will output v along channel c and then behave as P. In general, communications and channels can have any number of message fields. For example, $rec.a.b.m$ can represent a channel $rec.a$ carrying a message $b.m$.

The choice $P \,\Box\, Q$ offers the choice between processes P and Q. $P \,[\![A]\!]\, Q$ executes P and Q in parallel, where they must synchronise on all events in the set A—this is how processes interact with each other. Processes may also be defined by means of recursive definitions.

Each CSP process description is identified with its set of traces: the sequences of events that it can perform. For example, the process $in?x \rightarrow out!x \rightarrow Stop$ has $\langle in.3, out.3 \rangle$ as a possible trace, but not $\langle in.3, out.5 \rangle$: it cannot output 5 after inputting 3.

A specification $S(tr)$ is a predicate on traces. A process P satisfies a specification, written P **sat** $S(tr)$, if all of its traces meet the predicate. Trace specifications are used to capture safety requirements on processes: they require that all executions should be of a particular form, and hence that no execution should violate the specification.

2.2 Embedding CSP into PVS

A PVS ascii syntax is provided for the CSP operators described above (among others). Since we are interested in safety properties of security protocols, a CSP process is represented by its trace semantics in PVS. That is, given a fixed set, Σ, of all possible events, a process is represented by the set of traces of events from Σ that could be observed at the process interface. Recursive definitions are also possible by defining a monotonic functional, `H`, and then using the fixed point operator, `mu`, to generate its least fixed point, `mu(H)`. A type-correctness condition (TCC) is generated by PVS requiring a proof of `H`'s monotonicity. This is straightforward as all the CSP operators described above are monotonic with respect to their operand processes.

PVS support is provided in the form of theory files for general CSP trace semantics as follows:

- Traces
- Fixed point theory
- CSP operators
- specification and the **sat** relation

2.3 Analysing security protocols

A protocol is described in CSP in terms of the activity required of the participating agents, which may include servers and trusted third parties as well as parties that want to communicate. They are described in terms of the messages they send along *trans* channels, and receive along *rec* channels, and the manipulations they carry out on those messages. For example, the protocol which has user A signing a nonce challenge with a signature key s_A would be described by

$$USER_A \quad = \quad rec.A?j?n \rightarrow trans.A!j!\{n\}_{s_A} \rightarrow Stop$$

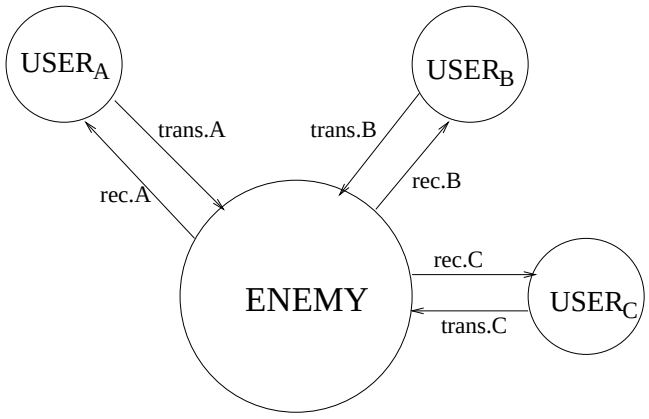


Figure 1: Communication in a hostile environment—the Dolev-Yao model

In this description, $USER_A$ receives a nonce n, apparently from j, on its receive channel $rec.A$. It responds by transmitting $\{n\}_{s_A}$ back to j along its transmit channel $trans.A$, and finishes. We use $\{m\}_k$ to denote message m encrypted with key k.

To analyse the protocol in the context of a possibly hostile environment, we use the Dolev-Yao model of a network [3]. In this framework the set of agent processes communicate with each other only by passing messages on the *trans* and *rec* channels to an 'enemy' that is in full control of the communications medium. He has the potential to block, re-direct, duplicate or fake messages on the medium. When cooperating with a user, the enemy accepts messages on the agent's transmitting channel *trans* and passes the message to the appropriate agent's receiving channel *rec*. This architecture is depicted in Figure 1.

To capture the enemy's ability to produce new messages, a 'generates' relation, $\vdash$, is defined allowing a new message, m, to be generated from a set of messages, S, already known. This is written as $S \vdash m$. This model of communication is defined in CSP by allowing agents to communicate with the enemy using transmission and reception events only: $trans.i.j.m$ is interpreted as agent i attempting to send message m to agent j, and $rec.i.j.m$ means agent i receives message m apparently from agent j. The enemy is defined as a CSP process that is willing to synchronise with all agents on *trans* and *rec* events. Any message received by the enemy (a *trans* event) is added to his set of knowledge. Any message the enemy intends to send (a *rec* event) must have been generated from his set of knowledge using the relation $\vdash$. This is defined in the PVS embedding by the process `enemy`, defined recursively by the function F: $\texttt{enemy(S)} = \texttt{F(enemy)(S)}$

```
F(X)(S) : process =
        (Choice! i, j, m : trans(i, j, m) >> X(add(m, S)))
     \/ (Choice! i, j, (m | S |- m) : rec(i, j, m) >> X(S))

enemy : [set[message] -> process] = mu(F)
```

In [10], the network is defined as a CSP process, NET which is the parallel

combination (synchronising on *trans* and *rec*) of the enemy together with all of the protocol agents and their possible communication partners. In PVS this is expressed as follows:

```
network : process =
        Par(trans? OR rec?)(enemy(INIT), Interleave(USER))
```

where `USER : [Identity -> process]` is the function defining the behaviour of all agents running the protocol, and `INIT` is the enemy's initial knowledge set.

Having defined the protocol in the worst possible environment, it is now possible to analyse the traces of this system and investigate whether certain properties relating to authentication hold.

2.4 Authentication properties

We take a message-based approach to authentication in order to enable a formal analysis. We say that one set of messages T authenticates another set of messages R if occurrence of a message in T guarantees that some message in R previously occurred. This is easily expressed as a property on traces: if a trace contains a message in the set T, then it should also contain a previous message in the set R.

By choosing appropriate T and R, we can express properties on the entire system NET that give various flavours of authentication. For example, if $T = \{rec.B.A.m\}$ and $R = \{trans.A.B.m\}$, then for NET to meet this property it must be the case that whenever $USER_B$ receives message m apparently from A, then $USER_A$ did indeed earlier transmit this message to user B. Clearly this will not be true for arbitrary messages, since if the enemy can generate m then he can pass it to $USER_B$ as if it had come from user A. So an authentication protocol designed to provide this property will have to design the authenticating message m appropriately.

A significant body of theory [10] has been developed on top of CSP trace theory for verifying properties of this type. This is based around finding a *rank function*, which assigns an integer value to each possible message, in such a way that all messages that might ever appear in a protocol run have positive rank, and only messages that can never appear have non-positive rank. Thus the enemy can only ever generate positive rank messages if it only ever has positive rank messages; and protocol agents must be shown to preserve positive rank, never introducing non-positive rank messages if they are never provided with them. If all occurrences of R within the system are blocked, then we aim to find a rank function where all messages in T have rank 0, establishing that they cannot occur in the restricted system. Thus for any T to occur, and R must occur previously.

This body of CSP theory has also been provided and verified within PVS [4], in the following PVS theories:

- authentication properties
- rank function properties
- enemy definition

- network definition and authentication theorem
- event datatypes
- rules for restricted parallel combinations (implemented from [10])
- rules for maintaining positive rank (implemented from [10])

The way PVS is used in practise is to find rank functions (where they exist) by carrying out a proof for the protocol in question providing only a 'blank' rank function, and then reducing all of the proof obligations on the correctness of the protocol to requirements on the rank function. These requirements often point the way to construction of a rank function, or alternatively they might be shown to be contradictory if the protocol is flawed, and might point the way to an attack.

The entire body of theory described thus far does not provide any general theoretical framework for handling time in the protocol descriptions or in the properties the system should meet. The contribution of this paper is the extension of the framework to include time.

3 Event-based Time

Time can be introduced into CSP in a number of ways. The most natural from the theoretical point of view would be to use the approach of Timed CSP [12], which is a mature modelling language in its own right. This approach integrates real-time in the form of the real numbers into the CSP language, and results in sophisticated but complex semantic model which would require much of the PVS framework to be redeveloped.

The other main approach (which is preferred in model-checking approaches because of the discrete nature of time) is to introduce a new special event *tock* into the alphabet of all processes. The resulting language is called *tock*-CSP. The event *tock* is used to represent the passage of one unit of time, and must be synchronised on by all processes in the system to reflect the fact that time passes at the same rate in all processes. Delays are introduced by requiring a number of *tock*'s to occur, and other time-sensitive behaviour such as timeout is also easily modelled in this framework. For example a fragment of a process which is repeatedly awaiting input but which will timeout and retransmit a value v if input is not received within one time unit might be described as

$$P(v) \quad = \quad in?x \rightarrow Q(x) \;\Box\; tock \rightarrow out!v \rightarrow P(v)$$

A detailed account of event-based time with illustrative examples can be found in [12].

By using an explicit event to mark the passage of time it is easier to integrate the handling of time into the existing PVS framework for CSP, which is based purely on events.

The modelling of time using this special event introduces some features that must be treated carefully. Care must be taken when defining processes in a timed environment. A process with no *tock* events does not mean that it is indifferent to the passage of time, but rather that it does not allow the passage of time. In our setting, all agents must synchronise on *tock* and none of them should

have the power to impede time. Therefore, process definitions that prevent *tock* events suggest a flaw in the network model.

The *urgent* events of a process are events that occur before any *tock* event is possible. (They are urgent because they must be performed immediately). It is essential that urgent events are not blocked: an uncooperative environment that blocks an urgent event would prevent the occurrence of any *tock* events. When using CSP to describe a process, it is therefore essential to include sufficient *tock* events to ensure such blocking does not occur, and that only events which should be urgent are indeed modelled in this way.

For example, the one-pass copy process described in CSP as

$$OPC \quad = \quad in?x \rightarrow out!x \rightarrow Stop$$

does not allow any time to pass at all, since it has no *tock* events in its description. Since *in* should not be urgent, it should allow any number of *tock*'s to occur before *in*. If *out* is also not urgent, then the appropriate description in *tock*-CSP would be:

$$\begin{aligned} TOPC &= in?x \rightarrow TOPC(x) \\ &\quad \Box\ tock \rightarrow TOPC \\ TOPC(x) &= out!x \rightarrow RUN_{tock} \\ &\quad \Box\ tock \rightarrow TOPC(x) \end{aligned}$$

Conversely, if the message should be passed on as soon as it is received (so *out* is urgent), then the description should be

$$\begin{aligned} TOPC' &= in?x \rightarrow out!x \rightarrow RUN_{tock} \\ &\quad \Box\ tock \rightarrow TOPC' \end{aligned}$$

Once *in* occurs, then *out* must occur before any more time passes. It is essential that the rest of the system does not block *out*, or else the model will contain a timestop state.

The translation mechanism, Ψ, from [12] provides a systematic way of translating CSP processes P into *tock*-CSP processes $\Psi(P)$, so that all of the events are non-urgent. The resulting process can either perform one of the enabled events, or else it can perform a *tock* event and remain in the same state. For example, $\Psi(OPC) = TOPC$. This mechanism allows processes without time-critical behaviour to be described in CSP and then translated naturally into *tock*-CSP.

Adding event-based time to the PVS embedding of CSP causes no problems because all of the theories concerning this embedding are parameterised by the event type. Only two changes were made to the original set of theories defining the embedding. Firstly, a `tock` constructor was added to the abstract data-type defining the CSP events in PVS. Secondly, a PVS definition of the *RUN* process (as used in the translation mechanism above) was added allowing it to be viewed as a primitive process in a timed setting in the same way that *Stop* is viewed in an untimed setting.

3.1 The Timed Network Model

We retain the Dolev-Yao model of the network in which the enemy is in full control of the communications medium, and the agents continue to communicate with each other via the enemy. The revised enemy is simply the original enemy with time added by means of Ψ: the most general enemy has no time-critical behaviour.

However, protocol agents' behaviour can be sensitive to time for a number of reasons:

- The values they produce (e.g. timestamps) can depend on the current time, and so the agents will have to be described explicitly in *tock*-CSP rather than as untimed CSP descriptions translated through Ψ. In fact they will need to keep track of the current time, and increment it on every *tock*.
- The response to a particular message might depend on the relationship between the current time and a time value within the message (usually to check that it is recent enough).
- The implementation of the protocol might include time-dependent behaviour such as timeouts or explicit delays.

In such cases, the CSP description of the protocol cannot be given purely as an untimed process translated through Ψ. Instead the timed behaviour will have to be described explicitly as a *tock*-CSP process. Section 4 gives an example of how this is done.

Furthermore, time can be introduced into the authentication properties that need to be checked. It may be necessary to check that a message received was in fact sent relatively recently; or perhaps that an entire protocol run has taken no more than a certain amount of time. If timestamps appear in the messages then such properties can be expressed within the existing framework of using one set of messages T to authenticate another set R. For example, if m is a message and l is a timestamp, then $\{rec.B.A.m.l\}$ might be used to authenticate $\{trans.A.B.m.l' \mid l - d \leqslant l' \leqslant l\}$: that the message was sent with a timestamp l' between $l - d$ and l.

Since the central rank function theorem remains true in the timed framework (as has been proved in PVS), this means that the existing rank function approach can be applied to verify that a time-sensitive protocol satisfies a timed authentication property expressed in this way.

4 An Example Analysis

The Wide-Mouthed Frog protocol is a simple protocol that uses timestamps. Its aim is to send a session key from one agent to another, via a server, using shared key cryptography. Timestamps indicate how recently a message was sent. The informal definition (taken from [2]) is stated as follows:

(1) $A \rightarrow S \ : \ A, \{B, Ta, Kab\}_{Kas}$

(2) $S \rightarrow B \ : \ \{A, Ts, Kab)\}_{Kbs}$

where Ta and Ts are timestamps, Kas is the key that A shares with S, Kbs is B's shared key, and Kab is a new session key generated by A. Both the server, S, and the receiver of message 2, B in this case, check that the timestamps lie within a specified range. If either message is too old then the session key is ignored.

From this description we gather information about the types of messages involved, the capabilities of the enemy, and the definition of the agents.

4.1 The Agents

In general, an agent i can either initiate a protocol run with another agent (via the server), or he can respond to an agent who chooses to initiate a run of the protocol with i. In this case, the word 'respond' is somewhat misleading because the agent receiving the message in a protocol run does not give any response. However, we shall continue to use it to distinguish between the roles of the agents and to be consistent with the terminology of examples in the literature. The server does not behave like an initiator or a responder and, therefore, requires a separate process definition.

We allow an agent to initiate and respond to arbitrarily many runs of the protocol. *UINIT* and *URESP* describe an initiator and a responder run respectively. The numeric arguments of `SERVER`, `UINIT`, and `URESP` represent the current time - i.e. we can view the agents as having synchronised clocks. These values are incremented for each *tock* event.

The definitions of `UINIT` and `URESP` are straightforward. Each deals with one non-urgent message, and so follows the style of the Ψ translation in ensuring that *tock* is always possible. For example, $UINUT(i,j,k)(l)$ representing a user i at time l wishing to send key k to user j (via the server S) is defined in CSP as follows:

$$\begin{aligned} UINIT(i,j,k)(l) \quad = \quad & trans.i!S!(i,\{j.l.k\}_{k_{is}}) \rightarrow RUN_{tock} \\ & \Box\ tock \rightarrow UINIT(i,j,k)(l+1) \end{aligned}$$

This is expressed in PVS explicitly as the fixed point of the function `FUINIT`. The responder is expressed in a similar style. We use `E(k,m)` to denote in ascii form the encrypted message $\{m\}_k$.

```
Y : VAR [nat -> process[event]]

FUINIT(i, j, k)(Y)(l) : process =
  (trans(i, s, conc(user(i),
   E(shared(i), conc3(user(j), time(l), session(k)))))
    >> RUN(tock))
 \/
  (tock >> Y(l + 1))

UINIT(i, j, k) : [nat -> process] = mu(FUINIT(i, j, k))

FURESP(i)(Y)(l) : process =
  (Choice! j, k, (l1 : nat | l >= l1 AND l1 >= l - d) :
   (rec(i, s, E(shared(i), conc3(user(j), time(l1), session(k))))
```

```
    >> (RUN(tock))))
 \/
  (tock >> Y(l + 1))

URESP(i) : [nat -> process] = mu(FURESP(i))
```

Note that in `FURESP` we use PVS's dependent type capability to capture the notion a 'recent' message: the responding agent is willing to accept any message (apparently from the server `s`) whose timestamp value `l1` lies between the current time `l` and the time `l - d` for a delay constant `d`.

The definition for `SERVER` requires the response to the receipt of a message to be urgent, and hence prevents a `tock` from occurring until the `trans` event has occurred. The server also performs the same check as the responder: that the timestamp `l1` on the received message is no older than `d` units of time.

```
FSERVER(Y)(l) : process =
 (Choice! i, j, k, (l1 : nat | l >= l1 AND l1 >= l - d):
  (rec(s, i, conc(user(i),
      E(shared(i), conc3(user(j), time(l1), session(k))))) >>
   (trans(s, j, E(shared(j), conc3(user(i), time(l), session(k))))
    >> (RUN(tock)))))
 \/
  (tock >> Y(l + 1))

SERVER : [nat -> process] = mu(FSERVER)
```

It is appropriate to make the `trans` event urgent because the server has control over when the messages are sent. The enemy never blocks *trans* events, so no timestamp will occur. On the other hand, `rec` events should not be urgent, since this would imply that the agent controls the time that messages must be received.

4.2 The Authentication Property

The property that we shall analyse is the following: if agent `b` 'responds' to a message containing timestamp value `t` then the protocol run was initiated at or after a time `t - d`, where `d` is the delay constant. Note that `t` would be the timestamp generated by the server. This is achieved by defining the sets `T` and `R` as:

```
T : set[mevent] =
     { e | e = rec(b, s, E(Kbs, conc3(A, time(t), Kab))) }

R : set[event] =
     { e | EXISTS (l : nat | t >= l AND l >= t - d) :
       e = trans(a, s, conc(A, E(Kas, conc3(B, time(l), Kab)))) }
```

The theorem that we want to prove is:

```
 authenticated : THEOREM network(enemy(INIT), USER) |> auth(T, R)
```

Now we have everything we need to perform the analysis except a definition for the rank function, `rho`, if one exists. In the usual way, we shall initially

declare `rho` as an uninterpreted function, and then use the PVS theorem prover to extract the conditions that `rho` must satisfy. It is from these conditions that we can either build a rank function to complete the proof, or show, because of a contradiction in the conditions, that no rank function exists (indicating the possibility of an attack).

In fact, in attempting to construct a rank function, we find that the following derived condition is required:

```
authenticated.3.1.2.1.2.2.2.4 :

[-1]  t >= l1
[-2]  l1 >= t - d
  |-------
[1]   l1 = t
```

This is true only when `d` is `0`. The problem is caused by the similarity in the messages that the server receives and the subsequent messages he transmits.

In fact, when $d > 0$ we can prove that the conditions required of the rank function are contradictory, and so we can conclude that no rank function can be constructed. From these derived conditions, it is possible to see the attack:

$$A \rightarrow S : A, \{B, Ta, Kab\}_{Kas}$$

$$S \rightarrow I(B) : \{A, Ta + d, Kab\}_{Kbs}$$

$$I(B) \rightarrow S : B, \{A, Ta + d, Kab\}_{Kbs}$$

$$S \rightarrow I(A) : \{B, Ta + 2d, Kab\}_{Kas}$$

$$S \rightarrow I(B) : A, \{B, Ta + 2d, Kab\}_{Kas}$$

$$S \rightarrow B : \{A, Ta + 3d, Kab\}_{Kbs}$$

The enemy can intercept a message sent by the server and, after prefixing the appropriate user identity to the message, send it back to the server as an initiator message. He can repeat this more than once providing that each interception occurs within the time delay, `d`. The effect of this is to keep the message's timestamp recent even though the age of the message could be more than the acceptable delay, `d`. This delaying tactic makes it possible for `b` to receive the message containing the key `kab` with timestamp `Ta+3d` (i.e. the event in `T`) even if the events in `R` (i.e. `a` transmitting with a timestamp between `Ta+2d` and `Ta+3d`) are blocked, since the original transmission with timestamp `Ta` is not blocked. If, as is implied by our problematic subgoals, `d` is set to zero then the problem does not arise because the enemy would have to act immediately and the message would not age. However, setting `d` to be zero would be an unrealistic assumption.

The alternative solution to this problem is to add an extra field to the encrypted part of the messages to distinguish between the initiator and responder messages. That is, the modified protocol becomes:

(1) $A \rightarrow S \ : \ A, \{B, Ta, Kab, initiate\}_{Kas}$

(2) $S \rightarrow B \ : \ \{A, Ts, Kab, respond\}_{Kbs}$

The corresponding modification to our PVS definition confirms this: re-running the proof easily yields a rank function which successfully verifies the amended protocol.

5 Discussion

Introducing time

This report has documented the generalisation of the PVS embedding of CSP (presented in [4]) by implementing event-based time. This allows the analysis of protocols that use timestamps in order to fulfil one or more security properties. In our example, the intended property is the freshness of a session key. However, it is still possible to analyse security protocols that have no time critical features, and existing proofs of authentication properties for untimed protocols have been repeated in the timed setting without modification.

The strength of the framework is that it will also allow more complicated time-dependent behaviour of the protocol agents to be expressed naturally, and analysed. Protocols, when modelled for analysis, are generally (implicitly) considered to halt if they do not make progress. Yet their implementations may use a timeout mechanism which triggers retransmission of a message a number of times before eventually giving up on the protocol. Such mechanisms might have a bearing on protocol correctness, particularly where timestamps are involved, and a framework for describing and analysing them will be of benefit. Investigation of such examples is the subject of ongoing research.

Other theorem-proving and model-checking approaches such as those described in [5, 1, 13, 6] among others do not offer this benefit explicitly, though it should be possible to extend their work to a general framework for handling time-critical behaviour of protocols, for example by modelling timeouts by use of timeout events (which may or may not contain specific time values). However, the coding up required would make the complex behaviour difficult to understand, and it is preferable to use a language designed to express such real-time behaviour.

Lowe's Casper tool [5] allows timed authentication properties to be checked by including a separate clock process within the model and requiring that every event involving time should synchronise with the clock to ensure the correct value. Thus protocol agents are not themselves described as timed processes, but they treat time values as they do other data values and use the clock as the time server. Similarly, Paulson defines the timestamp provided on a message in terms of a function of the events that have already occurred, which is akin to Lowe's approach, in the sense that the time value is obtained from some source external to the protocol. Again, time values are considered as data values, and time-critical behaviour such as timeouts are not expressed within this framework. Strand spaces [13] can handle timestamps, but would need times associated with all messages (even those without timestamps) in order to handle time-critical agent behaviour. The NRL Protocol Analyser does not currently handle timestamps, though they can (and will) be introduced by means of coding up what is essentially a global clock [7].

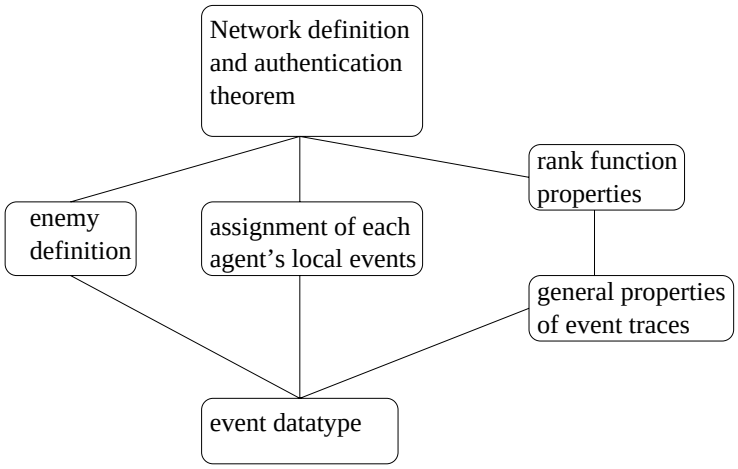
PVS

The PVS type system is very expressive and makes theory definitions more succinct. We have used subtyping to generalise the set of events in our CSP embedding. Events are now either value passing (`trans` and `rec` are events that enable the exchange of messages) or pure synchronisations (such as `tock`

events). Our example analysis has demonstrated the use of dependent types in process definitions. Recall that messages containing a timestamp were received only if their timestamp values were 'recent' according to a numeric argument representing the current time.

Altering the features of an existing PVS theory can be a nontrivial, time consuming exercise. Changes that are made to theories at the lower levels of a theory hierarchy can have a detrimental effect on the established proofs of theorems higher up the proof chain. The use of PVS's `status-proof` commands helps to indicate which proofs are affected. If, after the change, a lemma's status becomes `unfinished` then one can deduce that the change has had a direct effect on the lemma. Otherwise, if its status becomes `proved-incomplete` then there is an indirect effect on the lemma (the proof still works but refers to other `unfinished` lemmas). In our case, the addition of *tock* events has an (indirect) impact on the PVS implementation of the central authentication theorem. However, this work on the underlying theory needs to be done only once, and the payoff is a more general approach framework for analysing more classes of authentication protocol. In particular, a PVS proof shows that the central rank function theorem remains true in the more general semantic framework.

The following diagram gives an approximation of the hierarchy of the theories that are influenced by the modification of the event data-type. This is just a small part of the total hierarchy because the theories that occur lower down (such as the definitions of the CSP trace semantics) are unaffected by changes to `event`. (In fact, they are polymorphic with respect to events.)



The PVS environment also provides some useful features that have assisted the development of this project. In particular, the proof status commands are a source of information.

The theorem prover is used in an analysis to derive the conditions that must be satisfied by a rank function prior to its construction. These conditions are the leaves of the open branches of the proof tree. Our analysis has shown that a set of conditions can be inconsistent, signifying that no such rank function can be constructed. We have also shown that once a rank function has been defined it can be used to close the branches of the proof tree. The PVS strategy `GRIND` can be used to close such branches. However, it is common for `GRIND` to

generate multiple subgoals from one condition, and closing these branches can be very tedious.

Future Work

Investigation of this framework will benefit from a more significant example in which protocol agents exhibit time-critical behaviour. This does not require analysis of new protocols, but rather analysis of existing protocols in which the real-time behaviour of agents such as retransmission on timeout is modelled explicitly, rather than abstracted away. We need to understand the impact these implementation issues can have on protocol correctness, and may be able to obtain some general results about timing mechanisms.

As a tool for protocol analysis, this approach is not user friendly. The whole approach could benefit from a Casper-style interface [5] that allows a refined level of interaction. The construction of more specialised strategies that hide individual steps of a proof whilst allowing the user to intervene when some security or rank function expertise is needed. This automates the process and makes it more amenable to those with limited PVS knowledge by allowing communication at a level higher than primitive prover commands.

Acknowledgements

The authors are grateful to the UK EPSRC for funding under grant GR/M33402.

References

[1] G. Bella and L.C. Paulson. Using Isabelle to prove properties of the kerberos authentication system. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[2] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8, 1989.

[3] D. Dolev and A. C. Yao. On the security of public key protocols. In *IEEE Transactions on Information Theory*, volume 29(2), 1983.

[4] B. Dutertre and S. Schneider. Embedding CSP in PVS. an application to authentication protocols. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[5] G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, 1997.

[6] C. Meadows. Language generation and verification in the NRL Protocol Analyzer. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, 1996.

[7] C. Meadows. personal communication, 2000.

[8] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[9] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994.

[10] S. A. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 1998.

[11] S.A. Schneider. Timewise refinement for communicating processes. *Science of Computer Programming*, 28, 1997.

[12] S.A. Schneider. *Concurrent and Real-time Systems*. Wiley, 1999.

[13] J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 1999.