From Security Protocols to Systems Security: Making a case for systems security modeling[+]

Brian Monahan
Trusted Systems Laboratory
HP Laboratories Bristol
HPL-2003-147
August 4th , 2003*

E-mail: brian.monahan@hp.com

security protocols, systems security, SSML

When applying information security, we need to go beyond the analysis of individual security protocols and consider how they are used within distributed systems, software applications and services. Important as they are, security protocols only form a part of the overall security engineering design for a particular distributed system. The effective use of any security protocol will typically depend upon certain structural data such as key information being *available* for use by some - and at the same time made unavailable to others. Systems need to be designed with requirements like these in mind.

* Internal Accession Date Only Approved for External Publication
This report is the extended version of a paper entitled, 'From Security Protocols to Systems Security,' presented at the 11th International Cambridge Workshop on Security Protocols, Sidney Sussex College, Cambridge, UK. 2-4th April, 2003

From Security Protocols to Systems Security

Making a case for systems security modelling

Brian Monahan (brian.monahan@hp.com)

Trusted E-Services Lab, HP Laboratories
Filton Road, Bristol, BS34 8QZ, UK

July 2003

Abstract. When applying information security, we need to go beyond the analysis of individual security protocols and consider how they are used within distributed systems, software applications and services. Important as they are, security protocols only form a part of the overall security engineering design for a particular distributed system. The effective use of any security protocol will typically depend upon certain structural data such as key information being *available* for use by some - and at the same time made unavailable to others. Systems need to be designed with requirements like these in mind ([1–4]).

1 Introduction

When applying information security, we need to go beyond the analysis of individual security protocols and consider how they are used within distributed systems, software applications and services. Important as they are, security protocols only form a part of the overall security engineering design for a particular distributed system. The effective use of any security protocol will typically depend upon certain structural data such as key information being *available* for use by some - and at the same time made unavailable to others. Systems need to be designed with requirements like these in mind ([1–4]).

This report[1] describes work-in-progress that suggests extending the theory and analysis of security protocols towards a modelling approach for describing, analysing and algebraically simulating security aspects of systems.

We would argue that this involves using a software-supportable notation specifically designed for use by security trained professionals to design and explore the security aspects of systems. To this end, we are investigating an experimental notation, called here Systems Security Modelling Language (or SSML) to embody these systems designs. A variety of tools could be contemplated for investigating the security consequences of these designs within particular usage scenarios, to check resistance to particular attacks and to formulate specific trace and state properties which capture more precisely the security-related aspects of a system design.

Such an approach allows the system design, together with its security-related properties, to be iteratively evolved and pragmatically developed as a series of descriptions. The purpose of this evolution is to establish by pragmatic means a certain level of insight and confidence in the resulting final description. At that stage, it should by then be much clearer what the security-related properties of the design have to be and how each part contributes to establishing the security goals of a system. By evolution of a design, we mean a process of exploration and extension of designs, each of which better capture intention and functionality.

In addition to the system's security goals, the design development should have established what the initial assumptions are and have firmed up what structural data needs to be available at each processing stage. Even so, it is only entities contributing somehow to the security aspects of the system that need be modelled - the design needs only to be *sufficient* to establish the security case. Other entities that don't affect or somehow influence the system's security case need not be included in the model. In fact, once some entity has been shown to be irrelevant to the security case, the model could be simplified by removing that entity. Thus, security designs tend not to be completely definitive in every respect, but must be sufficient to capture the overall security situation that the system will find itself within.

[1] This report is the extended version of a paper [5] presented at the 11th International Cambridge Workshop on Security Protocols, held at Sidney Sussex College, Cambridge, on 2nd – 4th April, 2003.

To recap, we gain some "lightweight" assurance, via the tools supporting the security-modelling notation, that the evolved design achieves certain overall goals and functionality relevant to security. To gain a stronger degree of assurance and confidence in the design, it may be necessary to formally describe the *kernel* of a system in terms of mathematical logic and then formally prove that the most important security-related properties hold for that kernel.

However, constructing these formal specifications and proofs is a subtle and complex task, requiring significant logic-modelling skills. Furthermore, this will only be typically attempted for a stable system kernel where a highly assured verification of security and correctness is required. As the design has already been captured in the SSML modelling framework and has hopefully been matured into something already having a high level of confidence, this could provide an excellent starting point from which to extract a logically precise formal specification of a suitable system kernel. This formally represented version of the system kernel provides the basis for formally showing the security-related properties claimed for it, perhaps using tools for theorem-proving [6, 7] or symbolic model-checking [8].

Most of the remainder of this report discusses aspects of an experimental security-modelling language and accompanying tools by means of a short example. Before that, we briefly look at protocol descriptions (i.e Message Sequence Charts) and then consider approaches to the security analysis of protocols.

1.1 Protocol description notation – Message Sequence Charts

Security protocols have typically been described using the Message Sequence Chart (MSC) notation. For example, Fig 1 contains a classical MSC description of the well-known Needham-Schroeder public key authentication protocol.

```
A --> B : {A, Na}Pub(B)

B --> A : {Na, Nb}Pub(A)

A --> B : {Nb}Pub(B)
```

Fig. 1. Excerpt from the Needham-Schroeder Public Key Authentication Protocol

This intuitively conveys what the message flow is and broadly what each principal needs to do at each stage of the protocol. However, as recognised by Abadi in [9], this does not adequately document certain important aspects of the protocol such as:

- The goals of the protocol and what each participant expects to be jointly achieved.
- The initial assumptions made by each principal.
- What parts of each message were freshly generated and by whom.

Lowe showed in [10] that the particular protocol illustrated above is possible by essentially allowing the attacker to be an 'insider' (i.e. have the same access to authentication keys as the honest principles). This permits the attacker to perform a classic middle-person attack. However, as pointed out in [11], the original protocol design assumed that attacks would be mounted by external agents. This nicely illustrates how flaws in complex systems can arise, even for the best designed systems - a subsystem intended for one set of circumstances may eventually get used in another set of circumstances unintended or foreseen by the designers, leading to inappropriate behaviour. This further suggests that complex, distributed systems may need to be adaptive or evolutionary in their design, if they are to be successful in a changing environment.

The variation of a protocols' initial assumptions may easily lead to different attacks, and this was illustrated in [12] using an automated protocol analysis tool. This emphasises how security in complex, distributed systems depends upon context and the manner in which protocols are deployed and used.

1.2 Approaches to security protocol analysis

There are currently two main approaches to the attack and analysis of security protocols. Both of these approaches start from the classic Needham-Schroeder paper [1] on authentication in large networks. This landmark paper contains many of the basic principles and underlying issues of security protocol analysis from which all modern work has flowed.

Probability and complexity-based analysis In the probabilistic and complexity-theoretic approach favoured by cryptographers (e.g. [13–16]), the attacker tries to exploit protocol interactions in some way so as to reveal covert information about the underlying crypto primitives - the objective is to break the crypto primitives by subversion or by extracting key information. Whichever way is taken, if successful, the attacker would achieve full control.

The outcome of such an analysis is typically some estimate of the *work factor* required by the attacker in order to achieve their goal. This work factor may be the probability of extracting useful information, expressed in terms of a function of a given number of trials (i.e. attack runs) or it may be the number of trials required to extract particular information. A typical feature of this type of analysis has been the use of the Random Oracle assumption in which hash functions are modelled as functions randomly chosen from a uniformly distribution.

Algebraic and formal logic-based analysis In the algebraic and formal logic-based approach, begun by Dolev and Yao [17], and continued by many researchers e.g. [18, 7, 19, 20, 9, 21–28], the attacker tries to gain *reward* or *advantage* by playing different protocol players off against other protocol players, each of which may be using differing protocols. This usually involves some deceptive use of an honest principals role, where the attacker can exploit some inherent ambiguity within the protocols. Relative to given keys, the attacker can analyse and synthesise arbitrary messages, based upon what the attacker explicitly knows and subsequently uncovers. However, the attacker is only permitted to do cryptographic operations (e.g. encryption, signing) using those keys it has got *explicit* access to.

Process algebraic and trace theoretical approaches to protocol security have played a fundamental role in defining many important security concepts [29, 9, 24, 7, 30, 31, 28, 32–42], such as determinism, non-interference and correspondence properties. Security-related properties via typing within the Spi calculus framework have been investigated in [43, 29, 44–46]. The Multiset Rewriting approach is another emerging formalism for cryptographic protocols [22, 47, 48]. A good recent tutorial survey and overview for this entire research area maybe found in the proceedings of FoSAD 2000 [49–54].

Reconciliation of approaches Abadi and Rogaway began in [55] to draw these approaches more together, emphasising that assumptions of each model have corresponding assumptions in the other. Further work in combining these approaches has been actively pursued by Pfitzmann and Waidner [56, 57].

It is also known, however, that the analytic techniques dealing with security protocols mentioned above have their limitations. For example, it is widely acknowledged that the BAN logic approach does not adequately capture causation and emphasises what is known (*believed*) by the principles involved. Equally, there are digital signature schemes that have been proven secure within the Random Oracle approach, but where every implementation is equally insecure in their sense [15]. Proofs of security for a protocol only typically give *partial* assurance concerning integrity.

It is clear that currently no *single* approach to security protocol analysis has yet emerged that adequately captures the full richness and diversity of the way that systems are used and the security threats they face. This suggests that further work is needed to combine the efforts across several approaches when analysing and assessing systems security.

2 The classic Web service example

We now give the example we use to motivate the security-modelling notation. This example is a simplified variant of the well-known "single sign-on" problem, depicted in the form of a Web service. In the informal description, there are three distinct phases of operation:

Registration: Client registers and joins the Web Service, by interacting with the Registration Service.

Start a new session with Web Service: Client has to prove their membership in order to initiate a long-lived, but interruptible, session with the Web Service. The Client is given a time-limited token (a cookie) that allows them to rejoin the session later on.

Continue an existing session with Web Service: Client continues an existing session by sending the cookie back to the Web Service. If the cookie has time-expired, the session is closed.

The example given in Fig 2 contains a number of ambiguities and undefined notions that are typical of requirements descriptions. By their nature, requirements descriptions are necessarily informal - they contain the "germ" of the need to be fulfilled by some (designed) solution. The act of design itself involves imposing some order and structure to create a space of potential solutions, some of which can hopefully fulfil the spirit of the requirement that was originally expressed informally.

This example was chosen because it is somewhat typical of an applied service occurring naturally in its "raw" state, rather than something pre-packaged that is already well understood by the security protocols community, such as Kerberos.

A partial answer to that question is to provide ways for exploring and investigating security designs. Our approach is to capture the security aspects of a design in terms of a model that can be analysed in an effective way, supported by software tools. This approach involves capturing the *transactional behaviour* of the design in terms of structured data, events and so on. The purpose of any design modelling activity is to gain insight and establish confidence in the content of the design. It is important to try and keep the model as simple as possible - while retaining its essential relevance.

3 SSML - a notation for security modelling

Our notation, System Security Modelling Language, allows us to define entities representing systems in terms of their behaviour. Although SSML is superficially close in appearance to well-known programming languages (such as Java and Standard ML), its semantics does owe something to the process calculus tradition of CSP and CCS. Appendix A contains more details on the current syntax of SSML. Further discussion of the semantics underlying SSML is beyond the scope of this paper.

The intention is to capture aspects of the structural features of a security design together with pertinent aspects of dynamic behaviour. Because we are focussing here on security aspects, we choose to emphasise certain details and de-emphasise others. For example, we de-emphasise fine-grained conditional branching and case analysis, but focus instead upon the use of keys, encryption and so on. Such aspects, although de-emphasised for security analysis purposes here, are of considerable interest and importance, but at other stages of systems design and development.

For lack of space in this report, we will not present a full account of either the notation and its associated tools or of the complete example. This is left to form the subject of further publication.

There are three main groups of entity that make up an SSML description. These are:

1. **Contexts and types:** These entities represent structured data types and are also used to represent the state-space of each principal item that makes up a system.
2. **Events, methods and functions:** Each event is located at an instance of some context representing a principal and typically consists of straight-line code with no branching i.e. neither conditionals nor case analysis. Events can optionally receive an input, send an output, do both or even neither. Methods generally represent state-modifying operations that may yield a result. Pure functions can be specified algebraically in terms of conditional equations.

Let S be a Web service offering some online services over the Internet. Each potential customer C must firstly register their membership to be able to access this service in future via a registration service RS.

Following registration, C gains access to the service by fully proving their membership at the beginning of a session. Customer C then receives a time-limited token (a “cookie”) that permits re-admittance to S quickly without going through the full access dialogue. Once logged in, C has a set of access privileges allowing them to retrieve and/or update information.

The appropriate informal use-cases are diagrammed here:

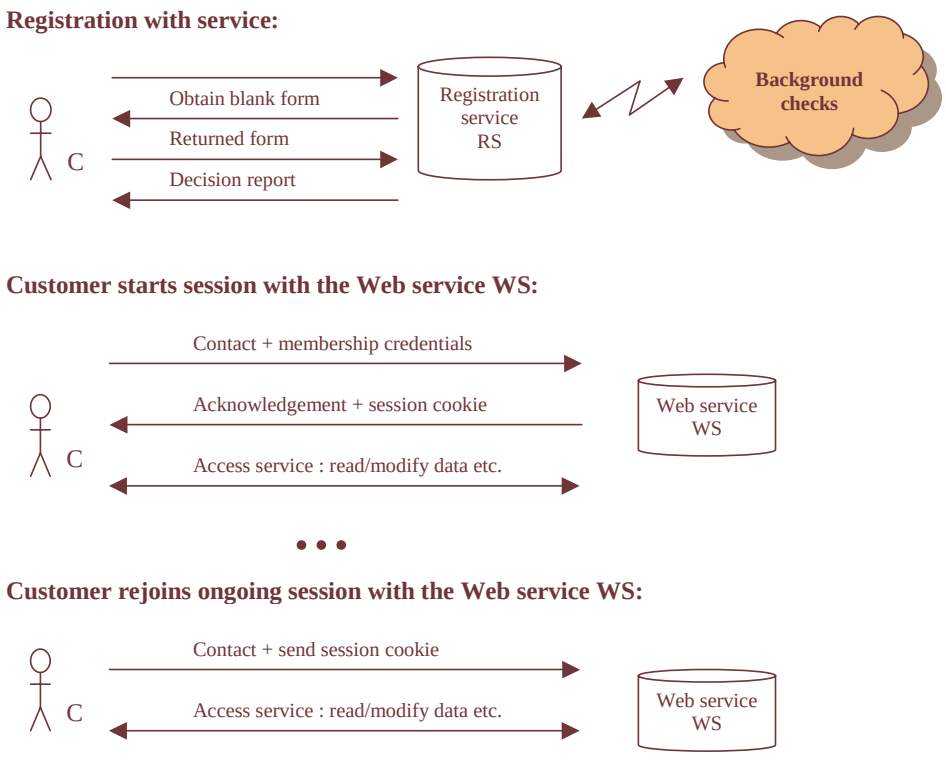

Fig. 2. Informal description of the Web Service

3. **Traces and system behaviours:** Traces are finite sequences of events, the message exchanges that are accepted and the state transitions that arise as a result. A system is defined in terms of (accepting) finite sets of traces.

We now move to discussing our Web Service example and illustrating the features mentioned above. In our design, the state-space for the Client, the Registration Service and the Web Service themselves will be represented in terms of appropriate *contexts*. Associated with these contexts will be a number of located events that represent some part of the interaction with another principal. These events process the input and output messages between principals. The overall *system behaviours* linking each of the Client's, the Registration Service and the Web Service are then defined in terms of traces.

We continue below with a more detailed description of the three main elements of SSML: contexts, events and systems behaviours.

3.1 Contexts

Describing complex systems involves structured data, and rather than involve full-blown objects and their semantic complexity (e.g. inheritance, overriding, etc.), we introduce **contexts** to represent this. A context is a collection of named, typed entities. An example is given in Fig 3 to define the Client's context.

```
context
  Client  =  {
    fixed
      visible  String   name;       // Client's name (not an Identity)

      visible  Asymkey  custK;      // Client's public key
      secret   Asymkey  privCustK;  // Client's private key

    volatile
      internal RegForm form;        // Registration form
      internal Nonce   n;           // A nonce value
      internal Cookie  ck;          // A "cookie" value, for Web Service

    persistent
      internal Identity registerID; // Registration Service Identity
      internal Identity serviceID;  // Web Service Identity

      internal Uint mem  = undef;   // Membership Number

    constraint
        (custK keypair privCustK);  // Key relationship
  }

type RegForm = unspecified; type Cookie = unspecified;
```

Fig. 3. Defining the Client's context

The context consists of a number of named fields, followed by an (optional) constraint part. Each field has a couple of classifiers. The first classifier is one of **fixed**, **volatile** or **persistent** - which briefly indicates how the field is used and initialised (or not). The second classifier is one of **visible**, **internal** or **secret**, and this determines security visibility characteristics. These classifiers are explained further in Appendix B. Each field is strongly typed - the types `Identity`, `Asymkey`, `Nonce`, `String` and `Uint` are built-in (there are others). The types `String` and `Uint` are conventional and stand for the type of strings and unsigned integers, as you would expect. The other types represent security-relevant *roles*.

For example, values of type Identity represent "standard" names for entities - in practice this might be a string containing a URI or some other systematically defined moniker. If A is a well-typed value, then

ident A is a value of type Identity. To be clear, Identity values are plaintext address-like references - they have no inherent security characteristics. We show later how these identity values can be used to access useful publicly visible information - such as public keys - bound to these identifiers. Values of type Asymkey and Nonce represent asymmetric keys and nonces respectively. The user-defined type RegForm is not further specified but is intended to represent a Registration Form of some kind (typically implemented in terms of XML or HTML). The user-defined type Cookie represents cookie values that are exchanged between Client and the Web Service and is not further specified. The order in which material is introduced is not significant - in general, there should be no cyclic dependencies between definitions.

Finally, note how the constraint part states the relationship between the customer's public key custK and their private key privCustK in terms of the binary key-pairing relationship, **keypair**. In general, this relation is regarded as being effectively intractable - even though it is classically definable. This relation is statically restricted to occurrences within verification predicates like the constraint part here - it is not permitted elsewhere.

3.2 Events

The dynamic behaviour of a security design is given in terms of events. Basically, an event represents a small chunk of state-dependant behaviour, and can input or output messages using the anonymous broadcast communication model, discussed further in Appendix B. Events are then strung together to form traces - where sets of traces describe the dynamic behaviour of a system.

A simple illustrative example of an event is given in Fig 4. This event is located at instances of a

```
event
  [Client A] acceptMembership {
    Identity regID, rID, cID;
    Binary encData;
    Nonce nA;

    input ["registered", regID, encData];

    [nA, rID, cID, A.mem] =
      decrypt ( encData ) with A.privCustK;

    check nA == A.n  &  rID == regID  &  cID == ident A;
  }
```

Fig. 4. Defining the acceptMembership event for the Client.

Client context - as specified by the "[Client A]" part. This effectively allows the event access to all of the components of its context state, here an instance of Client, during its execution. What the above says is that an input message can be accepted if it has the specified "shape" - it looks like: ["registered", regID, encData] This message has three components - the first being the string literal "registered", the second is an Identity value assignable to regID and the third is a Binary value assignable to encData. The effect of a successful pattern-match is to match the string literal and to overwrite the two variables.

The second statement says that, using Client A's private key privCustK, we can correctly decrypt the encData Binary value to give a message containing four elements [nA, rID, cID, A.mem], where nA is a Nonce value, rID and cID are both Identity values and A.mem is a field of type Uint associated with Client A (go back to Fig 3 for the definition of mem). Any previous values in the variables are ignored and overwritten if the whole statement successfully matches, with failure otherwise. Thus, partially successful matching does not cause any effective corruption.

Finally, the check statement checks the logical predicate to see if it is true - with failure if not. In this case, we check that:

1. The encrypted nonce nA is the same as A.n, a nonce sent previously and retained by Client A.

2. The identity rID is the same as regID, the identity quoted unencrypted in the main message.
3. The identity cID is the same as ident A, the identity corresponding to this Client.

An event can have at most one input statement and at most one output statement - typically, an event will only use either an input or an output.

```
event
  [StateContext ST] eventName {
    statements;

    input message;

    statements;

    output message;
  }
```

Fig. 5. General form of an event.

The general form of an event is given in Fig 5. In effect, events are (partial) state functions that may (optionally) input or output messages, while performing a constrained side effect upon the specified state context. If an event inputs a message, this may be accepted, refused or rejected. Rejection means failure, whereas refusal means the message could be accepted by some other event. Events are given by simple straight-line pieces of code, having no recursion, no loops, no case/switch statements and no (explicit) conditional branching via if statements familiar from programming languages like C and Java.

Typically, the input statement acts like a guard, allowing the selection of events that match the current message - they are different from the check statements mentioned earlier that are used to reject, thus yielding an (unrecoverable) failure. Other statement forms available include traditional assignment in both conventional and pattern-matching variants. Data operations in the form of pure functions can also be specified by conditional equations, and this includes recursive definition and conditional evaluation. However, we do not encourage this level of detailed description, since the intention is not to completely define all functional aspects of a system's behaviour, but to just focus on matters impinging upon security.

```
event
  [Client A] returnForm {
    Registry RX = access A.registerID as Registry;

    String data = userinput;
    String pw   = userinput;

    A.form = fillInForm(data, pw);

    A.n = random;
    Binary encForm =
      encrypt ( [A.n, ident A, A.form] ) with RX.regK;

    output ["return", ident A, ident RX, encForm];
  }

function RegForm fillInForm (String data, String pw);
```

Fig. 6. Defining the returnForm event for the Client

We give a further example of an event in Fig 6. The returnForm event performs the act of filling in the form using the (abstract) function fillInForm with data acquired by userinput, a data source that indicates local interaction with a user. This takes the form data plus the user chosen password value, pw. Note that the password value is not retained for later use by the Client context and so the user will have to supply this value as required. The filled-in form is then embedded inside an encrypted message consisting of three elements - a randomised nonce value A.n, an Identity value representing self, and finally the form. The nonce value A.n is used here for two reasons - firstly, to introduce an ideal form of randomised encryption (i.e. prevention of codebook attacks) and secondly, it is retained in the Client A for checking freshness later on - as performed within the event acceptMembership in Fig 4.

This message is encrypted using a public key value (RX.regK) obtained by de-referencing the local proxy value RX of type Registry. The encrypted data is then output, together with identities representing the Client and the Registry.

The RX value mentioned above is defined by an access expression:

```
Registry RX = access A.registerID as Registry;
```

Local proxy values constructed in this way only contain the fixed, visible fields of the entity that its Identity refers to. All other fields are made undef. These access expressions could in practice be realised by a number of different underlying mechanisms, such as Web Page lookup via URL, or LDAP access, each having the same overall effect.

For completeness, we give the Registry context that represents the state-space for the Registration Service in Fig 7 below. Also introduced there is the type HiSecurityKey which represents asymmetric keys whose binary size is 2048 bits. In general, any base type like Asymkey can be qualified by size.

```
context
  DataBase  = unspecified;  /* content to be determined */
  RegForm   = unspecified;  /* content to be determined */
  Registry  = {
    fixed
      visible  String   name;           // Registry name

      visible  HiSecurityKey regK;      // Public key
      secret   HiSecurityKey privRegK;  // Private key

    volatile
      internal Identity contact;        // Contact identity
      internal RegForm  formData;       // Form data

    persistent
      internal DataBase db;             // Registration database

    constraint
      regK keypair privRegK;
  }

type HiSecurityKey  =  <2048 Bit> Asymkey;
```

Fig. 7. Defining the Registry context

As described here, these events may appear somewhat isolated and unrelated to each other. What will link events together are system behaviours - and these are described in the next section.

4 System behaviours

As mentioned earlier, we specify the behaviour of security systems, consisting of a number of different principals or players, in terms of sets of traces - where a trace consists of a finite sequence of located events. As an example of this, we define the (intended) system behaviour corresponding to the Registration Service in Fig 9.

```
system RegistrationService (Client C, Registry RS) {
  init C.registerID == ident RS;

  ( [C]  sendHello;
    [RS] receiveHelloSendForm;
    [C]  receiveForm;
    [C]  returnForm;
    [RS] acceptProcessForm;
    [C]  acceptMembership
  )
}
```

Fig. 8. System behaviour corresponding to the Registration Service

Corresponding to each sequence of events, there are also the sequences of messages that are output and subsequently input. Accordingly, the above is essentially a (partial) function from the participants (i.e. specified by the Client and Registry types) satisfying the init predicate to potentially accepting traces of the system. If we instantiate this by giving particular instances of the Client and Registry, we obtain a sequence of events - which may or may not succeed. Thus we recognise that there could be instances that, although having the above form, will nonetheless lead to non-acceptance (rejection) due to failure.

A system description like the above is considered to be (weakly) well-formed if there is at least one accepting trace for some choice of arguments, and strongly well-formed if there is at least one accepting trace for every possible choice of arguments. In effect, strong well-formedness says that the system's description forms a total function.

There are other useful trace operators in addition to sequential composition $(T_1; T_2)$, such as simple choice $(T_1|T_2)$ and concurrent merge $(T_1||T_2)$. These are useful in defining complex trace sequences used in system behaviours and in trace predicates (equivalence and containment).

In the next section, we briefly consider how designs in SSML could be analysed using static analysis concepts and ideas from algebraic simulation.

4.1 Analysing the design

The purpose of a security analysis of a design is to determine if the design achieves certain security goals. Such goals are typically stated in terms of high-level concepts such as confidentiality, authentication, non-repudiation, non-interference and so on. Although such concepts can be mathematically defined in terms of trace, state and correspondence properties (e.g. [33, 20, 9, 34, 28, 58, 35, 7, 19]) translating these into precise, verifiable statements that are pertinent to particular designs is not so straightforward.

For us, analysis involves two distinct phases:

– **Checking of static integrity:** The objective of this phase is to ensure that the design satisfies certain static integrity constraints - and if not, to determine diagnostic information aiding the user in fixing any violations. The analysis involves techniques such as type-checking the usage of data items and flow analysis/abstract interpretation to ensure that information flows are suitably constrained [59, 46, 60, 39, 61].

  Important aspects of this checking includes:

1. Checking that system designs can accept some traces - a system is malformed if it ends up rejecting every trace (i.e. it has no behaviour of interest). This involves checking that message flows are feasible - that for a message to be accepted as input by an event, it must have been output by a preceding event. Additionally, the volatile fields must have been initialised before they are used in each possible run. However, it is also clear that these trace acceptance conditions could well involve general constraint satisfaction and hence need theorem-proving support.
2. Calculating the static set of valuable items as used by the system - for example, these are entities that are declared secret or only transferred solely in an encrypted form and not deliberately exposed. More subtly, this also includes indirectly valuable items because they can be used to gain access to other material already known to be valuable.

   In this fashion, we can detect when a system design makes certain kinds of security exposures. In addition, the set of valuable items represents those items that could form targets for attackers and thus need to be protected.

– **Exploration of dynamic behaviour:** System behaviour are defined here by sets of traces – where the traces include state transitions and the messages sent and received. The objectives of this phase are to examine, explore and where possible verify the dynamic characteristics of a system in combination with an environment. For us, an environment is a system corresponding to "the rest of the system" (i.e. the context) in which the system of interest operates.

  In particular, the attacks come from the environment – a successful attack being any coordinated set of actions controlled by the environment that provokes the entire system (i.e. environment and system together) to enter a bad state or to perform an illegal act.

  This type of analysis can usefully be viewed as exploring whether the system or the environment "wins" a certain contest between them – the environment "wins" if it can obtain reward and advantage in some systematic way, and the system of interest "wins" if it always defeats the best efforts of the environment to cause it to fail - i.e. the system wins if it doesn't lose.

We intend to use algebraic-based simulation and manipulation techniques to examine this behaviour and investigate the security of certain combinations of environment and system.

The CAPSL Integrated Protocol Environment described in [62], has been developed by Jonathan Millen and his team at SRI. This appears to be currently one of the most extensively developed security protocol analysis tools. CAPSL[2] provides an algebraic notation and tools for protocol analysis, which are based upon an algebraic rewriting logic. Although CAPSL's protocol notation essentially provides an algebraic form of Message Sequence Charts, it also provides ways to precisely state the protocol goals and initial assumptions for each principal. CAPSL descriptions are firstly translated into an intermediate logic-based form called CIL (CAPSL Intermediate Language) [63] which is in turn related to the MultiSet Rewriting formalism [22, 48]. CIL is principally used to provide a canonical protocol representation, allowing CIL descriptions of protocols to be exchanged between a variety of algebraic formal analysis tools, such as PVS and Maude

The CAPSL translator tools are currently available for Unix platforms.

5 Conclusions and further work

The purpose of this paper is to motivate interest in security-focused systems analysis and the need for security modelling and simulation tools, using the example of a Web Service to illustrate the idea. Web services are also used as an example in [45].

There is considerable natural scepticism about unconditional security or correctness arguments that ignore the operating context or environment. Usually, a system design can only be shown "correct" (i.e. meets its correctness obligations) under some assumptions about its mode of use and context - and the same certainly holds of security-related properties. For this reason, exploring and studying the combination of systems operating within particular environments, from a security point of view, is of great

[2] CAPSL stands for Common Authentication Protocol Specification Language.

interest. In other contexts, similar process-focused modelling approaches have also been advocated [64, 65].

We should be prepared to view a much richer landscape in which a system in one context is acceptably secure and performs a useful functional role. However, when used as a subsystem in some other context, the same system behaves insecurely and fails to operate as expected. Thus, the task of assessing the security of a system is closely associated with assessing those classes of environments in which a system operates safely.

Concerning the present form of SSML, some basic support tools already exist and a GUI-based analysis tool is currently under development. The purpose of such tools are to (1) provide ways of capturing models focusing upon the security aspects and (2) to explore their behaviour within particular environments using algebraic simulation and static analysis techniques. We hope to include goal statements that describe security-related properties, in terms of trace, state and correspondence predicates, and allow user-directed exploration of what happens under certain initial conditions (i.e. “what-if” scenarios).

6 Acknowledgements

I thank my colleagues Adrian Baldwin, Liqun Chen, Jonathan Griffin, Antonio Lain, Simon Shiu and Mike Wray at HP Labs for their helpful remarks and comments relating to this work. In addition, I am grateful for conversations with various protocol workshop participants which directly led to a number of improvements.

References

1. R. Needham, M. Schroeder: Using encryption for authentication in large networks of computers. CACM **12** (1978) 993–999
2. Anderson, R.: Security Engineering. Wiley (2001)
3. N. Ferguson, B. Schneier: Practical Cryptography. Wiley (2003)
4. W.R. Cheswick, S.M. Bellovin, A.D. Rubin: Firewalls and Internet Security. 2nd edn. Professional Computing Series. Addison-Wesley (2003)
5. Monahan, B.: From Security Protocols to Systems Security. In: Proc. of 11th International Cambridge Workshop on Security Protocols (2003). LNCS, Springer (2003) to appear.
6. T. Nipkow, L. C. Paulson, M. Wenzel: Isabelle/HOL - A proof assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
7. Paulson, L.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security **6** (1998) 85–128
8. Huth, M.: Secure Communicating Systems. Cambridge (2001)
9. Abadi, M.: Security Protocols and their Properties. In: Foundations of Secure Computation (F.L. Bauer and R. Steinbrueggen, eds.). NATO Science Series, Marktoberdorf, Germany, IOS Press (2000) 39–60
10. Lowe, G.: An attack on the Needham-Schroeder public key authentication protocol. Information Processing Letters **56** (1995) 131–136
11. Gollmann, D.: What do we mean by Entity Authentication? In: Proc. IEEE Symposium on Security and Privacy 1996, IEEE Computer Society (1996)
12. Monahan, B.: Introducing ASPECT - a tool for checking protocol security. Technical Report HPL-2002-246, HP Labs (2002) `http://www.hpl.hp.com/techreports/2002/HPL-2002-246`.
13. M. Bellare, P. Rogaway: Entity authentication and key distribution. In: Advances in Cryptology - Crypto 93 (ed. D. Stinson). Volume 773 of LNCS., Springer (1993)
14. R. Cramer, V. Shoup: A practical public key cryptosystem provably secure against adaptive ciphertext attack. In: Proc. of Advances in Cryptology - Crypto 98. Volume 1462 of LNCS., Springer (1998) 13–25
15. R.Cannetti, O.Goldreich, S.Halevi: The random oracle methodology, revisited (preliminary version). In: Proc. 30th Annual ACM Symp. On Theory of Computing, Perugia, Italy, ACM Press (1998) 209–218
16. R. Canetti, H. Krawczyk: Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels. In: Proc. of Eurocrypt 2001. Volume 2045 of LNCS., Springer (2001)
17. D. Dolev, A. Yao: On the security of public key protocols. Technical Report STAN-CS-81-854, Dept. of Computer Science, Stanford University (1981) Also in *Transactions on Information Theory*, 29(2):198-208, 1983.
18. M. Burrows, M. Abadi, R. Needham: A logic of authentication. In: Proceedings of the Royal Society of London A. Volume 426., Royal Society (1989) 233–271 Also publ. (condensed) in ACM Transactions on Computer Systems, 8(1): 18-36, February 1990.
19. Paulson, L.C.: Inductive analysis of the Internet protocol TLS. ACM Transactions on Computer and System Security **2** (1999) 332–351
20. R. Anderson, R. Needham: Programming Satan's Computer. In: Computer Science Today. Volume LNCS vol 1000., Springer (1995) 426–441 `http://www.cl.cam.ac.uk/ftp/users/rja14/satan.ps.gz`.
21. H. Comon, V. Shmatikov: Is it possible to decide whether a cryptographic protocol is secure or not? To appear in Journal of Telecommunications and Information Technology (2002)
22. N. A. Durgin, P. D. Lincoln, J. C. Mitchell, Scedrov, A.: Undecidability of bounded security protocols. Proc. FLOC Workshop on Formal Methods in Security Protocols (1999)
23. Gollmann, D.: Mergers and Principals. In: Security Protocols, (ed. B. Christiansen et al.). Volume 2133 of LNCS., Springer (2001) 5–13
24. Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In: TACAS. Volume 1055 of LNCS., Springer-Verlag (1996) 147–166
25. P. Ryan, S. Schneider: Modelling and Analysis of Security Protocols. Addison-Wesley (2001)
26. M. Rusinowitch, M. Turuani: Protocol Insecurity with Finite Number of Sessions is NP-complete. In: Proc. 14th IEEE Computer Security Foundations Workshop, IEEE (2001) 174–187
27. J.K. Millen, Hai-Ping Ko: Narrowing Terminates for Encryption. In: 1996 IEEE Computer Society Computer Security Foundations Workshop (CSFW9), County Kerry, Ireland, IEEE Computer Society Press (1996) 39–45
28. F. J. Thayer Fábrega, J. C. Herzog, J. D. Guttman: Strand Spaces: Proving Security Protocols Correct. Journal of Computer Security **7** (1999) 191–230
29. M. Abadi, A. Gordon: A calculus for cryptographic protocols: the Spi Calculus. Technical Report SRC-149, DEC-SRC (1998)
30. Lowe, G.: Defining Information Flow. Technical Report 1999/3, Department of Mathematics and Computer Science, University of Leicester (1999)
31. A. W. Roscoe, M.H. Goldsmith: What is intransitive non-interference? In: Proc. of 1999 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press (1999)
32. P.Y.A. Ryan, S. Schneider: Process algebra and non-interference. In: Proc. 1999 IEEE Computer Security Foundations Workshop, Mordano, Italy, IEEE Press (1999)
33. Ryan, P.: A CSP formulation of non-interference. Cipher, IEEE Computer Society Press (1991) 19–27

34. Roscoe, A.W.: CSP and Determinism in Security Modelling. In: Proc. of 1995 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (1995) 114–127
35. T. Y. C. Woo, S. S. Lam: Verifying authentication protocols: Methodology and example. In: Proc. Int'l. Conference on Network Protocols. (1993)
36. R. Focardi, R. Gorreri, F. Martinelli: Secrecy in Security Protocols as Non Interference. In: DERA/RHUL Workshop on Secure Architectures and Information Flow (S. Schneider and P. Ryan ed.). Volume 32 of Electronic Notes in Theoretical Computer Science., Elsevier, (1999)
37. J. Millen, V. Shmatikov: Constraint solving for bounded process cryptographic protocol analysis. In: Proc. 8th ACM Conference on Computer and Communications Security, ACM (2001)
38. V. Cortier, J.K. Millen, H. Rueß: Proving Secrecy is Easy Enough. In: 14th IEEE Computer Security Foundations Workshop (CSFW'01), Novia Scotia, Canada, IEEE Computer Society Press (2001) 97–109
39. H. Comon-Lundh, V. Cortier: Security properties: two agents are sufficient. In: Proc. 12th ESOP'2003. Volume 2618 of LNCS., Warsaw, Poland, Springer (2003) 99–113
40. S. N. Foley: A Non-Functional Approach to System Integrity. IEEE Journal on Selected Areas in Communications (2003)
41. F.Martinelli: Analysis of security protocols as open systems. TCS **290** (2003) 1057–1106
42. F. Martinelli: Symbolic Partial Model Checking for Security Analysis. In: Proc of MMM-ACNS 2003. LNCS, St. Petersburg, Russia, Springer (2003)
43. M. Abadi: Secrecy by Typing in Security Protocols. Journal of the ACM **46** (1999) 749–786
44. A.D.Gordon, A. Jeffrey: Types and effects for asymmetric cryptographic protocols. In: Proc. 15th IEEE Computer Security Foundations Workshop (CSFW 2002), IEEE (2002) 77–91
45. A.D.Gordon, R. Pucella: Validating a web service security abstraction by typing. In: Proc. 2002 ACM Workshop on XML Security, Fairfax VA, USA, ACM Press (2002)
46. C. Bodei, P. Degano, H. R. Nielson, F. Nielson: Static analysis for secrecy and noninterference in networks of processes. In: Proc. PACT'01. Volume 2127 of LNCS., Springer (2001)
47. I. Cervesato, N. Durgin, J. C. Mitchell, P. Lincoln, A. Scedrov: Relating Strands and Multiset Rewriting for Security Protocol Analysis. In: Proc. 15th IEEE Computer Security Foundations Workshop, IEEE (2000) 35–51
48. I. Cervesato: A Specification Language for Crypto-Protocols based on Multiset Rewriting Dependent Types and Subsorting. In: Proc. of SAVE 2001, Paphos, Cyprus (2001) `http://www.disi.unige.it/person/DelzannoG/SAVE01/cer.ps.gz`.
49. P.Y.A. Ryan: Mathematical Models of Computer Security. In: FOSAD 2000. Volume 2171 of LNCS., Springer (2001) 1–62
50. P. Syverson, I. Cervesato: The Logic of Authentication Protocols. In: FOSAD 2000. Volume 2171 of LNCS., Springer (2001) 63–137
51. P. Samurati, S. de Capitani di Vimercati: Access Control: Policies, Models and Mechanism. In: FOSAD 2000. Volume 2171 of LNCS., Springer (2001) 137–196
52. J. D. Guttman: Security Goals: Packet Trajectories and Strand Spaces. In: FOSAD 2000. Volume 2171 of LNCS., Springer (2001) 197–261
53. A. D. Gordon: Notes on Nominal Calculi for Security and Mobility. In: FOSAD 2000. Volume 2171 of LNCS., Springer (2001) 262–330
54. R. Focardi, R. Gorreri: Classification of Security Properties (Part 1: Information Flow). In: FOSAD 2000. Volume 2171 of LNCS., Springer (2001) 331–396
55. M. Abadi, P. Rogaway: Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). Journal of Cryptology **15** (2002) 103–127
56. B. Pfitzmann, M. Waidner: A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In: Proc. of IEEE Symposium on Security and Privacy, Oakland, California, IEEE Computer Society Press (2001) 184–200
57. B. Pfitzmann, M. Schunter, M. Waidner: Cryptographic Security of Reactive Systems (Extended Abstract). Electronic Notes in Theoretical Computer Science **32** (2000)
58. J.D. Guttman, F.J.T.F.: Authentication tests and the structure of bundles. Theoretical Computer Science (2001)
59. F. Nielson, H.R. Nielson, C. Hankin: Principles of Program Analysis. Springer (1999)
60. D. Clark, C. Hankin, S. Hunt: Information flow for Algol-like languages. Journal of Computer Languages (2002)
61. A. Sabelfeld, A. C. Myers: LanguageBased InformationFlow Security. IEEE Journal on selected areas in Communications **21** (2003)
62. G. Denker, J.K. Millen, H. Rueß: The CAPSL Integrated Protocol Environment. Technical Report SRI-CSL-2000-02, SRI, Menlo Park, California, USA (2000)
63. G. Denker, J.Millen: CAPSL intermediate language. In: Proc. of FLoC Workshop on Formal Methods and Security Protocols. (1999)
64. Bruns, G.: Distributed Systems Analysis with CCS. Prentice-Hall (1997)
65. G. Birtwistle, C. Tofts: Relating Operational and Denotational Descriptions of Demos. Simulation Practice and Theory **5** (1997) 1–33
66. F. Stajano, R. J. Anderson: The Cocaine Auction Protocol: On the Power of Anonymous Broadcast. In: Proc. of 3rd Information Hiding Workshop, Dresden, Germany. LNCS, Springer (1999) `http://www-lce.eng.cam.ac.uk/ fms27/papers/cocaine.pdf`.

A Appendix: A Systems Security Modelling Language

SSML is an experimental notation for expressing systems security designs and their properties. This Appendix summarises and amplifies various details mentioned earlier in the paper. A simplified extract from the full grammar is included below.

Types, Contexts and Fields

Types are conventional in that they constrain the range of values that expressions can denote. Contexts consist of a collection of fields whose values may be constrained under some invariance . Examples can be found in Fig 3 and 8. Contexts are used to provide both structured data types and state-spaces that are associated with state-modifying events.

Each field is defined by two qualifiers, its type, its name and an optional initial value. The intention of these qualifiers is given in more detail in the tables below.

Events

Events are state-based partial functions that can input a message or output a message (or both or neither). They are associated with a context instance representing the state. On input, a message is read without commitment and could either be *accepted*, *refused* or *rejected*. *Guard statements* can be used to refuse messages, allowing another event to run. *Check statements* can cause failure and message rejection. Output messages are dispatched directly. Anonymous Broadcast is the communications model used and is further explained below.

System Behaviour as Sets of Traces

System behaviour is defined in terms of sets of traces - where a trace is a sequence of events, state transitions and accepted messages that are transferred between different partners. It is important to note that the traces do not fully define the behaviour of any principal, but just defines relevant interactions between principles.

Anonymous Broadcast Communications Model The Anonymous Broadcast communications model, originally introduced by Stajano and Anderson in [66], essentially says that messages are sent and received asynchronously, not involving a binary message handshake or rendezvous. The basic properties of message transfer are in this model:

- Once a message is sent, it could be received later by anyone,
- Once a message is received, it has already been sent, but by anyone.

In other words, addressing information is advisory, but cannot be guaranteed from a security point of view. Just because a message claims to have been sent by someone, it doesn't mean that they necessarily sent it. Equally, just because you send a message to someone, it doesn't necessarily mean that only they will see it.

We further assume that if Alice first sends m1, and then later sends m2, then if Bob receives both messages, then Bob will receive them in the order they were sent by Alice. This means that only "local" send ordering is preserved on receipt. Naturally, this does not imply any other global temporal ordering - only local temporal ordering. We also assume reliability of message sending in that any message sent can be received eventually. Because we make no restrictions on who can receive a message once sent, we assume implicitly that an "intended" recipient will see the message. From this we assume that attackers cannot actively prevent messages from being received by the intended recipient.

We finally observe that, from a process theoretic point of view, we can assimilate the Anonymous Broadcast model in terms of primitive handshaking by postulating a universal process (called the Internet) that accepts messages sent concurrently by processes, allowing then to be forwarded upon request to other such processes. Although not particularly challenging, this idea can be made formally precise in terms of well known process calculi such as CCS or CSP.

Partial grammar for System Security Modelling Language

```
context       ::=  'context' contextdefn_1 ... contextdefn_n
contextdefn   ::=  id '=' contextexpr
contextexpr   ::=  'unspecified' ';' | '{' items constraint '}'
constraint    ::=  empty | 'constraint' expr ';'

items         ::=  item_1 ... item_n
item          ::=  itemclass itemdefn_1 ... itemdefn_n
itemclass     ::=  'fixed' | 'volatile' | 'persistent'
itemdefn      ::=  accessmod type id initvalue ';'
accessmod     ::=  empty | 'visible' | 'internal' | 'secret'
initvalue     ::=  empty | '=' expr

event         ::=  'event' '[' statespec ']' id '{' inputpart outputpart '}'
inputpart     ::=  empty | stmts 'input' message ';'
outputpart    ::=  stmts output
output        ::=  empty | 'output' message ';'
stmts         ::=  empty | stmt_1 ';' ... ';' stmt_n ';'
stmt          ::=  vdeclstmt | patstmt | checkstmt | guardstmt | methodcall
patstmt       ::=  pattern '=' expr
checkstmt     ::=  'check' expr
guardstmt     ::=  'guard' expr
vdeclstmt     ::=  simptype id_1 ',' ... ',' id_n initvalue

system        ::=  'system' id '(' statespecs ')' '{' tracedefns sysdefn '}'
tracedefns    ::=  empty | 'trace' tracedefn_1 ... tracedefn_n
tracedefn     ::=  id '=' trace_expr ';'
sysdefn       ::=  initpred trace_expr acceptpred
initpred      ::=  empty | 'init' expr ';'
acceptpred    ::=  empty | 'accept' expr ';'
```

Lifetime qualifiers	
fixed	The variable's value is long-lived and once defined, remains effectively constant. Its value is not changed once defined (c.f. **static**, **final** in Java). It may be assumed to have been well-defined in some way prior to system runs.In practice, these might be changed/modified - but so infrequently when compared to the frequency of system runs. In practice, these might be changed/modified - but so infrequently when compared to the frequency of system runs.
volatile	The variable's value is short-lived and should be *freshly* (re-)initialised within each run of the system before first use. However, its value will persist from event to event inside a system run.
persistent	The variable's value is long-lived, it can be changed/modified within events. Like **fixed** variables, it may be assumed to have been well-defined in some way prior to system runs.

Accessibility qualifiers	
visible	The variable's value is visible to external observers. It is usually important that its value be highly available - it should be widely disseminated and published.
internal	The variable's value is regarded as internal and unavailable to external observers. Such a variable is only accessable from within a "live" executing event or method - it's static value is **undef**.
internal	The variable's value is regarded as internal and unavailable to external observers. Additionally, this value cannot be assigned, passed as an argument or used in any expression context, except by a specific crypto operation (e.g. encryption, signing). As for **internal** variables, such a variable is only accessable from within a "live" executing event or method - it's static value is **undef**.

Cryptographic Expressions	
encrypt $message$ **with** key	Encrypt the message with the given encryption key. We assume that labels indicating the appropriate algorithm and the cryptographic scheme (i.e. asymmetric vs symmetric) to use are all embedded within the key data.
decrypt $message$ **with** key	Decrypt the message with the given decryption key. We assume that labels indicating the appropriate algorithm and the cryptographic scheme (i.e. asymmetric vs symmetric) to use are all embedded within the key data.
sign $message$ **with** key	Construct a digital signature for the given data using the given signature key.
verify $(message, signature)$ **with** key	Verify the given digital signature for the given message against the given signature verification key.
key_1 **keypair** key_2	The key-pair relation for given keys, where the key data contains label information indicating the appropriate cryptographic scheme.
hash $message$	Standard cryptographic hash of given message.
hmac $message$ **with** key	Key-dependent cryptographic hash of message using a hashing key.