Project IST-1999-11583

**Malicious- and Accidental-Fault Tolerance
for Internet Applications**

Formal Model of Basic Concepts

André Adelsbach and Birgit Pfitzmann (Editors)
Universität des Saarlandes (D)

MAFTIA deliverable D4

Public document

May 28, 2003

Editors

André Adelsbach	*Universität des Saarlandes (D)*
Birgit Pfitzmann	*Universität des Saarlandes (D)*

Contributors

André Adelsbach	*Universität des Saarlandes (D)*
Sadie Creese	*DERA, Malvern (UK)*
Tom McCutcheon	*DSTL, Malvern (UK)*
Birgit Pfitzmann	*Universität des Saarlandes (D)*
Matthias Schunter	*Universität des Saarlandes (D)*
William Simmonds	*DERA, Malvern (UK)*
Michael Waidner	*IBM Zurich Research Lab (CH)*

Abstract

The overall goal of MAFTIA Workpackage 6 is to rigorously define the basic concepts developed by MAFTIA, and to verify results of the work on dependable middleware. The main objective of this first deliverable is to present a rigorous model of the most important concepts of malicious- and accidental-fault tolerance. Additionally, work has already started on the two later objectives of specification and verification with CSP, i.e., in a formal language and with a model checker, and on a sound combination of cryptographic and formal analysis techniques.

The main result is a general rigorous model for the security of reactive systems using a simulatability definition. It comprises various types of faults (attacks), synchrony and topology as considered in MAFTIA. A proof of secure message transmission, a low-level middleware protocol, is shown in this model. A composition theorem for this model and a relation to integrity requirements have been proven; they allow modular proofs in this model, which is essential for the modular design approach of MAFTIA. These theorems can also be seen as a step towards the combination with formal analysis because system specifications can be abstract (in particular deterministic) even if the implementations use cryptography, and thus specifications of higher layers can use standard tools.

The initial CSP investigations were carried out for synchronous contract signing. A distributed synchronous protocol of this type and complexity has not been formally modeled in the literature before. It should provide good links with an abstract treatment of cryptography in the future. The results include a general CSP model of a synchronous network, which was then specialized with malicious faults pertaining to contract signing. This model was used to obtain positive verification of the MAFTIA synchronous contract signing protocol.

Contents

1 Introduction

The project MAFTIA systematically investigates the tolerance paradigm for building dependable distributed systems. For this, it combines techniques and notions from fault tolerance and various areas of security, such as intrusion detection and cryptographic protocols. While Workpackage 1 concentrates on identifying and interrelating the basic concepts and providing a general framework and Workpackages 2 to 5 design concrete mechanisms and protocols within this framework, the goal of Workpackage 6 is to enable the assessment of the dependability achieved by rigorously defining the basic concepts and verifying results of the work on dependable middleware.

Such clarity of concepts and validation of designs is important even in an overall context where one assumes that perfection cannot be reached and thus tolerance is needed: The sources of the imperfection should not be the MAFTIA concepts, but come in automatically, e.g., by imperfection of physical and organizational measures intended to ensure the assumed independence of certain components. Even here a rigorous model helps because it forces the designers to be very clear about such assumptions.

1.1 Concepts to Formalize

The basic concepts to formalize in the area of MAFTIA middleware are the

- models of
  - faults (attacks),
  - synchrony, and
  - topology,
- and the security goals.

For verification, the main challenge lies in distributed protocols using cryptographic primitives.

There is a lot of previous literature on different aspects of this task. However, there was no general model that covers all the aspects needed in MAFTIA. For instance, the field of (non-cryptographic) distributed protocols provides models of interacting components with various kinds of synchrony, but in both the synchronous and asynchronous case we had to extend the model to obtain realistic timing in the presence of adversaries. The field of protocol verification provides formalizations of goals (e.g., temporal logic), but this does not cover general goals involving confidentiality aspects, as needed in security. Cryptography provides many definitions of individual goals of individual types of primitives or protocols, but no really general framework in which one can talk of different synchrony models or, in particular, of abstract goals as suitable for modular design and the link to formal methods. More related literature and motivation for the approaches chosen can be found in the individual chapters.

1.2 Rigorous and Formal Models in MAFTIA

A definition or proof can be rigorous or formal. "Rigorous" means that it is of the quality required in mathematics (and most mathematicians would use the word "formal" synonymously), i.e., in a mixture of text and mathematical symbols such that definitions are deemed unambiguous and proofs convincing to an expert reader. "Formal" means in a specific formal system (e.g., a logic), i.e., the exact syntax of definitions and the exact rules allowed in a proof are fixed.

The objectives of this workpackage were to

1. rigorously define important basic concepts of MAFTIA,

2. formalize some properties and protocols in the language CSP and verify them with a modelchecker,

3. and to investigate how cryptography can be integrated into such formalizations in a faithful way, although cryptography typically needs notions of probabilism, polynomial-time algorithms, and small error probabilities, which are not present in standard formal systems and not amenable to current proof tools.

The original goal of this deliverable (as expressed in the title) was to present results regarding Objective 1 only. However, as work on all three objectives has started and the work is interrelated, we decided to present a cross section through all three aspects of the work.

Chapter 2 addresses Objective 1 and Parts of Objective 3. This combination is useful because the rigorous definitions cannot abstract from cryptography. Hence they must include aspects like probabilism and polynomial-time algorithms. However, they are immediately formulated to make it as easy as possible to later use abstract specifications (i.e., without such complications) and to define what it means that such a specification is fulfilled in a cryptographic sense. More motivation for this is given in Section 1.3.

An example of such an abstract specification and the proof that a concrete cryptographic implementation fulfils it is given in Chapter 3, i.e., this chapter belongs to Objective 3 alone. The example used is one of the most basic primitives of the MAFTIA middleware, secure point-to-point channels in an open environment.

Chapter 4 addresses Objective 2 along a concrete example. The example, contract signing, is presented first in its original terms; then general conceptual aspects of the formalization are presented and finally the concrete formalization. The example was chosen to enhance discussion in view of Objective 3: An almost rigorous description was available in advance, only one type of cryptographic primitive is used, and no confidentiality requirements are made, so that the abstract goals can be formulated in a standard CSP way. The general results are a synchronous network model with a fairly general adversary; some rules for state-space reduction and models of termination.

In exchange for all these extensions, we do not present all model variants here. In particular, we concentrate on a synchronous model in all chapters. Work on asynchronous models is ongoing for all three objectives; for Objective 1 and 3 it has already been presented in [49].

1.3 Cryptography and Abstractions

Security proofs for systems involving cryptography are getting increasing attention in theory and practice, and they are used for increasingly large systems. While for some time most of the effort concentrated on primitives like encryption and signature schemes, or authentication and key exchange, currently work is under way on medium-sized systems like secure channels, payment systems, and anonymity systems. In the future, one might want to prove even larger systems like the entire MAFTIA middleware, electronic-commerce architectures or distributed operating systems that use cryptography.

Both the cryptographic and the formal-methods community are working on such proofs, and the techniques are quite disjoint. One of our goals is to link them to get the best overall results: proofs that allow abstraction and the use of formal methods, but retain a sound cryptographic semantics.

Abstracted Models. In the formal-methods community, one tries to use established specification techniques to specify requirements and actual protocols unambiguously and with a clear semantics. Moreover, most work aims at proofs that are at least automatically verifiable. To make this possible, cryptographic operations are almost always treated as an infinite term algebra where only predefined equations hold (in other terminology, the initial model of an abstract data type) as introduced in [20]. For instance, there is a pair of operators E_X and D_X for asymmetric en- and decryption with a key pair of a participant X. Two encryptions of a message m from a basic message space M do not yield another message from M, but the term $E_X(E_X(m))$. The equation $D_X(E_X(t)) = t$ for all terms t is defined, and the proofs rely on the abstraction that no equations hold unless they can be derived syntactically from the given ones. Early work using this approach for tool-supported proofs was rather specific, e.g., [41, 39]; nowadays most work is based on standard languages and general-purpose verification tools, as initiated, e.g., in [52, 37, 2].

A problem with these models is the lack of a link between the chosen abstractions and the real cryptographic primitives as defined and sometimes proven in cryptography. The main issue is not even that one will somehow need to weaken the statements to polynomial-time adversaries and allow error probabilities; the problem is that the cryptographic definitions say nothing about *all* equations. For instance, the accepted cryptographic definition of secure asymmetric encryption only requires that an adversary in a strong type of attack cannot find out anything about the message (see [9, 16]), but nothing about possible relations on the ciphertexts. One can construct examples, at least artificial ones, where proofs made with the abstractions go wrong with encryption schemes provably secure in the cryptographic sense [46].

Faithful Abstraction. The problem can be approached from both sides—cryptography can try to offer stronger primitives closer to the typical abstractions, or formal methods can be applied based on weaker abstractions that are easier to fulfil by actual cryptography. Both approaches presuppose that one defines what it means that some abstraction is fulfilled in a cryptographic sense. Both also need proofs that working with the abstractions leads to meaningful results in the real cryptographic sense, i.e., the abstractions should be faithful. This is illustrated in Figure 1.1. We will show how the theorems proven in Chapter 2 help in this program.

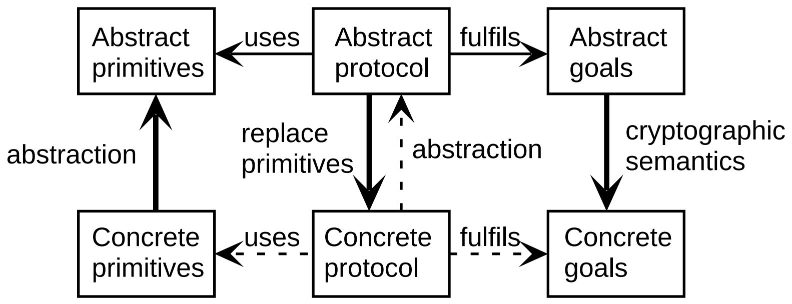

Figure 1.1: Goals of faithful abstraction. Bold arrows should be defined once and for all, normal arrows once per protocol. It should be proven that dashed arrows follow automatically.

What does Cryptography Gain from Abstractions? Cryptographers may ask why one should bother with abstractions: why not continue to make all proofs on the lower layer of Figure 1.1, i.e., as reduction proofs if asymmetric cryptography is involved? Indeed, abstractions and formal methods neither increase the expressiveness nor make the overall results more rigorous. However, one can hope that the specifications get nicer, the proofs shorter, and tool support easier. (The large number of papers using unfaithful abstractions indicates how alluring these arguments are.) Indeed rigorous cryptographic definitions are long (always involving details of the attack, error probabilities etc.), and many had to be strengthened later. Similarly, most proofs are currently only sketches, and some have contained serious gaps. This will get worse with larger systems; just imagine proving an entire electronic-commerce framework by a reduction only because it uses signatures in some places.

2 Security of Reactive Systems

In this chapter we present a general model for secure reactive systems. Reactive means systems where users may make many inputs and get many outputs at different times and in different places.

2.1 Overview of the Model

In Section 2.3, we present the general system model, i.e., components and their interaction, including possible attackers and an environment. In MAFTIA terms, this contains the synchrony model and the range of topologic models that can be covered.

Section 2.4 gives our central definition of security by a notion that one system simulates another one. Essentially, a system Sys_1 is considered *at least as secure as* a system Sys_2 if whatever any adversary A_1 can do to any honest user H in Sys_1, some adversary A_2 can do to the same user H in Sys_2 essentially with the same probability. System Sys_1 is often a real system using concrete cryptographic primitives, while Sys_2 is an ideal system, i.e., a specification. Thus our central definition is not one of specific security attributes like confidentiality. That would not be possible in the general case, i.e., for a wide range of primitives and applications as considered in MAFTIA, e.g., *what* should be confidential, under what conditions etc.? However, a simulatability definition essentially means that all properties that the ideal system has by construction are carried over to the real system. The ideal system is typically written in a centralized, non-cryptographic form, and thus, e.g., it is typically simple to specify that it just does not output some information to somebody. The fact that simulatability covers confidentiality aspects is also its main distinction from the notion that a real system implements a specification, as known from normal verification of distributed systems.

In Section 2.5, we specialize the system model to a class of standard systems. This corresponds to a specific class of fault models for standard topologies; the main model contains malicious faults of machines in arbitrary access structures and three classes of channels. This covers the main MAFTIA middleware systems. We also sketch crash failures. (An extension to adaptive attacks has been given in the asynchronous context in [49].)

In Section 2.6 we validate the model by investigating its relation to several natural variants; most are equivalent.

In Section 2.7 we define what it means for a system to provide almost arbitrary integrity properties in a cryptographic sense. We then prove that (a) proofs of such properties made for the ideal system also hold for the real system and (b) logic derivations among integrity properties are valid for the real system in the cryptographic sense. This is a partial validation of the claim that simulatability carries all the properties of the ideal system over to the real system.

In Section 2.8 we define the composition of systems in our model. We prove that the specification of one system can be used in the design of another system while preserving statements about the security of the overall systems. This is essential for using the simulatability approach in layered systems like the MAFTIA middleware.

The integrity statements and the designs of higher layers in compositions can be abstract and thus accessible to formal methods. This is one of the links between the cryptographic and the formal approach as desired for Objective 3 of this workpackage of MAFTIA.

2.2 Related Literature

Simulatability is the main approach in cryptography at providing general definitions. However, for a long time it was only worked out for *function evaluation* in contrast to reactive systems, i.e., all parties make one input at the beginning and get one output at the end [57, 25, 7, 40, 13]. For this case, [13] contains a composition theorem. Our Section 2.8 can be seen as an extension of this to reactive systems.

There are two main approaches at extending simulatability to reactive systems. The first, constructive, approach describes the ideal system as a global state-transition machine and requires the global state to be shared among all participants in the real system [29, 23]. This is not feasible or desirable in scenarios like secure channels and most other MAFTIA protocols, where many participants carry out many 2- or 3-party subprotocol runs at different times.

The second, descriptive, approach only considers the "outside" behaviour of the system. Brief sketches have been around for some time [47, 13]; more detailed definitions were made in [35, 36, 30] (and a preliminary version of our own model in [46]). For none of them, theorems about composition or the preservation of integrity properties have were given, hence this is an important new aspect of our work. The main advantage of [35, 36] is a formal language, π-calculus, as machine model. However, it lacks abstraction: the specifications in both papers essentially comprise the actual protocols and are specific to certain cryptographic primitives used. Hence tool support would also not be possible yet because even the specifications involve ad-hoc notation for generating random primes etc. (Combining some of their language techniques with our abstraction techniques looks promising.) In [30], a somewhat restricted class of systems is considered (straight-line programs and information-theoretic security) because their main goal was general constructions. Particular new aspects of our model are the precise timing models, a precise treatment of the interaction of users and adversaries, and a very general trust (fault) model. The simulatability approach has also been applied to specific reactive problems without a general definition [22, 8, 15].

Another approach to link abstract and cryptographic definitions was made in [4], but so far not for the reactive case: They show that equality of terms as in the original model from [20] is equivalent to computational indistinguishability, for symmetric encryption schemes with some special security properties. An approach to provide integrity properties with a cryptographic semantics was first made in [42], but not as rigorously as here and without a relation to simulatability definitions.

2.3 System Model

In this section, we present the system model underlying our security definitions. Components are represented as interacting automata. A typical model of a fault-tolerant system would represent at least those components that can fail independently as different automata. Thus automata with their connections may often coincide with topology in the standard sense, i.e., different computers and lines between them, but they may also represent different processes on one trusted computing basis that does not allow malicious processes to infect each other.

As mentioned in Section 1.2, we only present the definitions for a synchronous network model, although we also have asynchronous counterparts [49]. One advantage of a synchronous model is that the security definitions then also include the timing, so that security exposures via timing channels are captured. The physical assumptions underlying the synchronous model are discussed at the end of Section 2.3.2.

Section 2.3.1 defines the general system model, i.e., machines and how a collection of machines runs. Section 2.3.2 contains security-specific system aspects, in particular systems with users and adversaries.

2.3.1 General System Model

The machine model is probabilistic state-transition machines, similar to probabilistic I/O automata as sketched in [38]. For clarity, one particular notation is fixed. The semantics contains our extended timing model.

Definition 2.1 *(Machines and Ports)*

a) A name *is a string over a fixed alphabet* Σ*. A* port p *is a pair* $(name_p, dir_p)$ *of a name and a Boolean value called direction; we write* $name_p?$ *and* $name_p!$ *for in- and output ports, respectively. We write* p^c *for the* complement *of a port* p*, i.e.,* $name_p!^c := name_p?$ *and vice versa. For a set* P *of ports, let* $\mathsf{In}(P) := \{p \in P | dir_p = ?\}$ *denote the input ports and similarly* $\mathsf{Out}(P)$ *the output ports.*

b) A machine M *for a synchronous system is a tuple*

$$\mathsf{M} = (Ports_\mathsf{M}, States_\mathsf{M}, \delta_\mathsf{M}, Ini_\mathsf{M}, F_\mathsf{M})$$

of a finite set of ports, a subset of Σ^* *of states, a probabilistic state-transition function, and sets of initial and final states. The inputs are tuples* $I = (I_p)_{p \in \mathsf{In}(Ports_\mathsf{M})}$ *of one input* $I_p \in \Sigma^*$ *per input port, and the outputs analogous tuples* O*.* δ_M *maps each such* *pair (s, I) to a finite distribution over pairs (s', O). For a set $\hat{M}$ of machines,*[1] *let* $\mathsf{ports}(\hat{M}) := \bigcup_{\mathsf{M} \in \hat{M}} Ports_{\mathsf{M}}$.

c) *"Machine* M_1 *has machine* M_2 *as a (blackbox)* submachine*" means that it has the state-transition function as a blackbox. Hence* M_1 *can "clock"* M_2*, i.e., decide when to cause state transitions.*

d) *For computational aspects, a machine is regarded as implemented by a probabilistic interactive Turing machine [28], and each port by a read-only or write-only tape. Its complexity is measured in terms of the length of the initial state, represented as initial worktape content (often a security parameter).*[2] *Any time bounds are assumed to be explicitly given and efficiently computable.*

◇

Below, we distinguish correct machines, adversaries and users in particular in how they are clocked, because one cannot assume adversaries to adhere to synchronization rules. However, we first define more general collections of machines and clocking schemes because they will be useful in some proofs.

Definition 2.2 *(Collections)*

a) *A* collection $\hat{C}$ *is a finite set of machines with pairwise disjoint sets of ports. Each set of complementary ports $c = \{p, p^c\} \subseteq \mathsf{ports}(\hat{C})$ is called a* connection *and the set of these connections the* connection graph $\mathsf{G}(\hat{C})$.

b) *By* $\mathsf{free}(\hat{C})$ *we denote the* free *ports, i.e.,* $\mathsf{ports}(\hat{C}) \setminus \mathsf{ports}(\hat{C})^c$. *A collection is* closed *if* $\mathsf{free}(\hat{C}) = \emptyset$.

c) *A* clocking scheme *for $\hat{C}$ is a mapping $\kappa : \{1, \ldots, n\} \to \mathcal{P}(\hat{C})$ with $n \in \mathbb{N}$. Here $\mathcal{P}$ denotes the powerset. Intuitively, $\kappa(j)$ denotes the machines switched in Subround j of each round. We also write κ as a tuple $(\kappa_j)_{j=1,\ldots,n}$, and we will write Subround j of Round i as $[i.j]$.*

d) *A collection is called polynomial-time if all its machines are.*

◇

Definition 2.3 *(Runs) Let a collection $\hat{C}$ and a clocking scheme κ for it be given.*

[1]We mostly use a straight font for machines, functions and constants, and italics for sets and other variables, and add a hat to some sets for better distinction.

[2]Measuring it in terms of all inputs received so far would allow two machines called "polynomial-time" to carry out exponentially long computations, e.g., by doubling the message length in each round.

a) A run *(or "execution" or "trace") r is a function (a matrix) with the domain $\hat{C} \times \mathbb{N} \times \{1, \ldots, n\}$. Each element $r[\mathsf{M}, i, j]$ is either a quadruple (s, I, s', O) of the old state, inputs, new state, and outputs of machine M in subround $[i.j]$, or $\perp$ if $\mathsf{M} \notin \kappa(j)$ or $s \in F_{\mathsf{M}}$ (not clocked in this subround or final state).*

If all machines reach a final state, the run only consists of $\perp$ from then on. We then identify it with its prefix until the last round i where a machine switched.

b) For every tuple $ini \in Ini := \times_{\mathsf{M} \in \hat{C}} Ini_{\mathsf{M}}$ of initial states, a random variable of runs is defined inductively over the subrounds (in lexicographic order):

- *For each (M, i, j), the pair (s', O) in $r[\mathsf{M}, i, j] = (s, I, s', O)$ has the distribution $\delta_{\mathsf{M}}(s, I)$.*
- *Each resulting output $O_{\mathsf{p}!}$ is available as input at $\mathsf{p}?$ when the machine M' with port $\mathsf{p}?$ is clocked next, i.e., in the next subround $[i', j'] > [i, j]$ with $\mathsf{M}' \in \kappa(j')$. If several inputs arrive until that time, $I_{\mathsf{p}?}$ is their concatenation (earliest first).*

  For the computational case, $I_{\mathsf{p}?}$ is only an l-bit prefix of this value, where l is the run-time of M' for the initial value $ini_{\mathsf{M}'}$.[3]

c) The family of these random variables is written

$$run_{\hat{C}} = (run_{\hat{C},ini})_{ini \in Ini}.$$

d) For a number $l \in \mathbb{N}$ of rounds, l-round prefixes $run_{\hat{C},ini,l}$ of runs are defined in the obvious way. For a function $l : Ini \to \mathbb{N}$, this gives a family $run_{\hat{C},l} = (run_{\hat{C},ini,l(ini)})_{ini \in Ini}$.

$\diamond$

Definition 2.4 *(Views and Restrictions)*

a) The view of a subset $\hat{M}$ of a closed collection $\hat{C}$ in a run r is the restriction of r to $\hat{M} \times \mathbb{N} \times \{1, \ldots, n\}$. This gives a family of random variables

$$view_{\hat{C}}(\hat{M}) = (view_{\hat{C},ini}(\hat{M}))_{ini \in Ini},$$

and similarly for l-round prefixes.

If all machines in $\hat{M}$ reach a final state, the view only consists of $\perp$ from then on. We then identify it with its prefix until the last full round with another value.

[3]This prevents polynomial-time machines interacting with unrestricted machines from getting super-polynomial views.

b) *For a run r and a set $P \subseteq \mathsf{ports}(\hat{C})$ of ports, let $r\lceil_P\colon P \times \mathbb{N} \times \{1, \ldots, n\} \to \Sigma^*$ denote its restriction to these ports.*[4] *This notation is carried over to the random variables.*

   We also identify a trace with its prefix until the last full round with another value than $\bot$.

◇

2.3.2 Security-Specific System Model

Now we define specific collections for security purposes, first the system part and then the environment, i.e., users and adversaries. Typically, a cryptographic system is described by an intended structure, and the actual structures are derived using a trust model, see Section 2.5. However, as a wide range of trust models is possible (some others are sketched in Section 3.2 of [44]), it is useful to keep the remaining definitions independent of them as follows.

Definition 2.5 *(Structures and Systems) A* structure *is a pair $struc = (\hat{M}, S)$ where $\hat{M}$ is a collection of machines called* correct machines*, and $S \subseteq \mathsf{free}(\hat{M})$ is called* specified ports*. Let $\bar{S} := \mathsf{free}(\hat{M}) \setminus S$ and $\mathsf{forb}(\hat{M}, S) := \mathsf{ports}(\hat{M}) \cup \bar{S}^c$.*

A system *Sys is a set of structures. It is called polynomial-time if there is a polynomial that bounds the runtime of the machines of all structures (as a function of k).* ◇

The separation of the free ports into specified ports and others is an important feature of this particular reactive simulatability definition. The specified ports are those where a certain service is guaranteed. Typical examples of inputs at the specified ports are “send message m to *id*” for a message transmission system or “pay amount x to *id*” for a payment system. The ports in $\bar{S}$ are additionally available for the adversary. The ports in $\mathsf{forb}(\hat{M}, S)$ will therefore be forbidden or at least unusual for an honest user to have. In the simulatability definition below, only the events at specified ports have to be simulated one by one. This allows *abstract* specification of systems with *tolerable imperfections*. For instance, if the traffic pattern is not hidden (as in almost all cryptographic protocols for efficiency reasons), one can abstractly specify this by giving the adversary one busy-bit per message in transit in the ideal system. Even better, he should only get one busy-bit per subprotocol run (e.g., a payment) and the internal message pattern of the subprotocol should not tell him more. A detailed example is given in Chapter 3.

The following definition contains another important aspect: Both honest users and an adversary are modeled as stateful machines H and A apart from the system, see Figure 2.1. First, honest users should not be modeled as part of the machines in M because they are arbitrary, while the machines have prescribed programs. Secondly, they should not be

[4]This is well-defined as follows: Each port $p \in P$ belongs to either $\mathsf{In}(Ports_{\mathsf{M}})$ or $\mathsf{Out}(Ports_{\mathsf{M}})$ for exactly one machine $\mathsf{M} \in \hat{C}$. In the first case, $r\lceil_P[p, i, j] := I_p$ if $r[\mathsf{M}, i, j] = (s, I, s', O)$ and $\bot$ otherwise; and similarly in the second case.

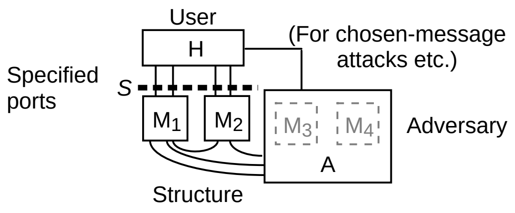

Figure 2.1: Configuration of a reactive system. The standard case $\mathit{Ports}_{\mathsf{H}} = S^c$ is shown. The gray part shows that a structure is typically derived from an intended one by omitting some machines that have been corrupted.

replaced by a quantifier over input sequences, because they may have arbitrary strategies which message to input next to the system after obtaining certain outputs. They may even be influenced in these choices by the adversary, e.g., in chosen-message attacks on a signature scheme; thus H and A may communicate. At least in the computational case, arbitrary strategies (i.e., adaptive attacks) are not known to be replaceable by arbitrary input sequences. Thirdly, honest users are not a natural part of the adversary because they are supposed to be protected from the adversary. In particular, they may have secrets and we want to define that the adversary learns nothing about those except what he learns "legitimately" from the system (this depends on the specification) or what the user tells him directly.

Definition 2.6 *(Configuration)*

a) A configuration *conf of a system Sys is a tuple* $(\hat{M}, S, \mathsf{H}, \mathsf{A})$ *where* $(\hat{M}, S) \in \mathit{Sys}$ *is a structure and* $\hat{C} := \hat{M} \cup \{\mathsf{H}, \mathsf{A}\}$ *a closed collection. The corresponding clocking scheme is* $\kappa = (\hat{M} \cup \{\mathsf{H}\}, \{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{A}\})$. *Runs and views are now given by Definitions 2.3 and 2.4.*

b) The set of configurations is written $\mathsf{Conf}(\mathit{Sys})$, *and those with polynomial-time user and adversary* $\mathsf{Conf}_{\mathsf{poly}}(\mathit{Sys})$. *"*poly*" is omitted if it is clear from the context.*

c) Typically, the initial states of all machines are only a security parameter k (in unary representation). Then one considers the families of runs and views restricted to the subset $\mathit{Ini}' := \{(1^k)_{\mathsf{M} \in \hat{C}} | k \in \mathbb{N}\}$ *of Ini, and writes* $\mathit{run}_{\mathit{conf}}$ *and* $\mathit{view}_{\mathit{conf}}(\hat{M})$ *for* $\mathit{run}_{\hat{C}}$ *and* $\mathit{view}_{\hat{C}}(\hat{M})$ *restricted to* Ini', *and similar for l-round prefixes. Furthermore,* Ini' *is identified with* $\mathbb{N}$; *hence one can write* $\mathit{run}_{\mathit{conf},k}$ *etc.*

◇

Clocking the adversary between the correct machines is the well-known model of "rushing adversaries". We also did not assume that the users (humans or application programs) are synchronized with the rounds of a particular protocol. The generality of the chosen clocking scheme is discussed further in Section 2.6.3.

This abstract model of synchrony is based on the following concrete assumptions:

Assumption 2.1 *(Synchrony) A synchronous machine reads inputs and makes outputs at locally fixed times. The clocks of correct machines are sufficiently synchronized during a system run, and network delays are sufficiently predictable, that any message output by a correct machine M_1 at its local output time of Round i has arrived at the local input time of Round $i+1$ (and after that of Round i) of the receiving machine M_2 if that is also correct. The period between (local) input and output time of one round is sufficient for the machine's computations of that round. Thus the round length depends on the initial inputs, typically the security parameter.*

The first assumption avoids that timing differences within a round leak, e.g., to prevent attacks as in [33]. The second one is the standard assumption that allows an implementation of rounds. Note that there is no assumption at all about the adversary's synchronization, i.e., he will clearly not be restricted to subrounds in the real world. The abstract subrounds just allow him to base his behaviour in one round on everybody else's behaviour in the same round, which is his optimal strategy.

2.4 Main Security Definition: Simulatability

In this section, we define security in the sense of simulatability for reactive systems, i.e., our basic notion of security as motivated in Section 2.1. Section 2.4.1 contains the actual definitions, Section 2.4.2 presents some important lemmas needed for proofs in this model.

2.4.1 Simulatability

In this section, we define what it means for one system Sys_1, typically a real system, to be as secure as another system Sys_2, typically a specification. In other words, we say that Sys_1 simulates Sys_2.

We only want to compare each structure of Sys_1 with certain corresponding structures in Sys_2. A typical case is shown in Section 2.5. Generally, an almost arbitrary mapping f is allowed as specification of "corresponding", only certain naming conventions are necessary. In particular, the users of one system should not automatically get port names forbidden in the other system:

Definition 2.7 *(Valid Mapping, Suitable Configuration)*

a) A function $f : Sys_1 \to \mathcal{P}(Sys_2)$ is called a valid mapping *for systems Sys_1 and Sys_2 iff*

$$p^c \in \mathsf{free}(\hat{M}_1) \Rightarrow p \notin \mathsf{forb}(\hat{M}_2, S_2) \quad \wedge \quad p^c \in S_2 \Rightarrow p \notin \mathsf{forb}(\hat{M}_1, S_1)$$

for all structures with $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$.

b) *Given Sys_2 and f, the set $\mathsf{Conf}^f(Sys_1) \subseteq \mathsf{Conf}(Sys_1)$ of* suitable *configurations contains all those configurations $(\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1)$ where H has no ports from $\mathsf{forb}(\hat{M}_2, S_2)$ for any $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$.*

◇

The restriction to suitable configurations $\mathsf{Conf}^f(Sys_1)$ serves two purposes in simulatability: First it excludes users that are incompatible with $(\hat{M}_2, S_2)$ simply because of name clashes. Secondly, it excludes that H connects to unspecified free ports of $(\hat{M}_2, S_2)$. This is necessary for the abstract specification of tolerable imperfections. Recall the example of an ideal system that gives the adversary one busy-bit per subprotocol run (Section 2.3.2): Clearly there is no such bit in the real system; we only need it to capture that whatever the adversary learns in the real system is not more than this bit. As we will require indistinguishability of the views of H, these unspecified ports must only be used by the adversary. The following lemma shows that with regard to one structure alone, the restriction to suitable configurations is without loss of generality.

Lemma 2.1 *For every configuration $conf_1 = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1) \setminus \mathsf{Conf}^f(Sys_1)$, there exists $conf_{\mathsf{f},1} = (\hat{M}_1, S_1, \mathsf{H_f}, \mathsf{A}_{\mathsf{f},1}) \in \mathsf{Conf}^f(Sys_1)$ such that $view_{conf_{\mathsf{f},1}}(\mathsf{H_f})$ equals $view_{conf_1}(\mathsf{H})$ except for port renaming.* □

Proof. We construct $\mathsf{H_f}$ by giving each port $p \in Ports_{\mathsf{H}} \cap \mathsf{forb}(\hat{M}_2, S_2)$ a new name.[5] As $conf_1$ is closed, the port p^c also occurs in it. The first condition on a valid mapping f implies $p^c \notin \mathsf{free}(\hat{M}_1)$. Hence p^c belongs to A_1 or H and we can indeed rename it. Then the runs and in particular the user's view are unchanged except for this renaming. ■

As the definition of computational indistinguishability (originally from [58]) is essential for the simulatability definition, we also present it here.

Definition 2.8 *(Indistinguishability) Two families $(\mathsf{var}_k)_{k\in\mathbb{N}}$ and $(\mathsf{var}'_k)_{k\in\mathbb{N}}$ of random variables (or probability distributions) on common domains $(D_k)_{k\in\mathbb{N}}$ are called*

a) *perfectly indistinguishable ("=") if for each k, the two distributions are identical;*

b) *statistically indistinguishable ("$\approx_{SMALL}$") for a class SMALL of functions from $\mathbb{N}$ to $\mathbb{R}_{\geq 0}$ if the distributions are discrete and their statistical distances*

$$\Delta(\mathsf{var}_k, \mathsf{var}'_k) := \frac{1}{2}\sum_{d\in D_k} |P(\mathsf{var}_k = d) - P(\mathsf{var}'_k = d)| \in SMALL$$

(as a function of k). SMALL should be closed under addition, and with a function g also contain every function $g' \leq g$. Typical classes are EXPSMALL containing all

[5]By *new name*, we always mean one that does not occur in the systems and configurations already under consideration. For this, we assume w.l.o.g. that no finite set of systems contains all port names over the given alphabet.

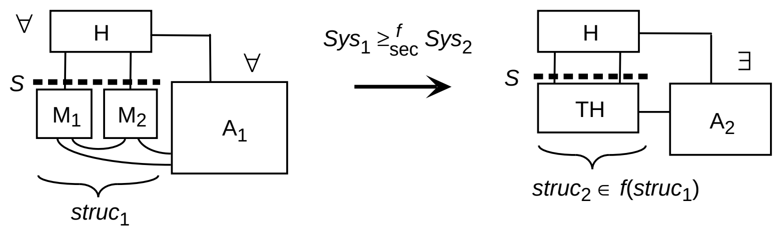

Figure 2.2: Example of simulatability for the typical case $S_1 = S_2 =: S$; the views of the user H are compared.

functions bounded by $Q(k) \cdot 2^{-k}$ for a polynomial Q, and the (larger) class NEGL as in Part c).

c) *computationally indistinguishable ("$\approx_{\mathsf{poly}}$") if for every algorithm* Dis *(the distinguisher) that is probabilistic polynomial-time in its first input,*

$$|P(\mathsf{Dis}(1^k, \mathsf{var}_k) = 1) - P(\mathsf{Dis}(1^k, \mathsf{var}'_k) = 1)| \leq \frac{1}{\mathsf{poly}(k)}.$$

(Intuitively, Dis*, given the security parameter and an element chosen according to either var_k or var'_k, tries to guess which distribution the element came from.) The notation $g(k) \leq 1/\mathsf{poly}(k)$, equivalently $g \in NEGL$, means that for all positive polynomials Q, $\exists k_0 \forall k \geq k_0 : g(k) \leq 1/Q(k)$.*

We write $\approx$ if we want to treat all cases together. ◇

The following definition captures that whatever an adversary can achieve in the real system against certain honest users, another adversary can achieve against the same honest users in the ideal system. This is illustrated in Figure 2.2. (Alternatives, e.g., adding an output to the adversary, will be discussed in Section 2.6.)

Definition 2.9 *(Simulatability) Let systems Sys_1 and Sys_2 with a valid mapping f be given.*

a) *We say $Sys_1 \geq_{\mathsf{sec}}^{f,\mathsf{perf}} Sys_2$ (*perfectly at least as secure as *Sys_2 for f) if for every suitable configuration $conf_1 := (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^f(Sys_1)$, there exists a configuration $conf_2 := (\hat{M}_2, S_2, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ with $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ (and the same H) such that*

$$view_{conf_1}(\mathsf{H}) = view_{conf_2}(\mathsf{H}).$$

b) *We say $Sys_1 \geq_{\mathsf{sec}}^{f,SMALL} Sys_2$ (*statistically at least as secure as*) for a class SMALL if the same as in a) holds with statistical indistinguishability of all families $view_{conf_1,l}(\mathsf{H})$ and $view_{conf_2,l}(\mathsf{H})$ of l-round prefixes of the views for polynomials l.*

c) *We say* $Sys_1 \geq_{\mathsf{sec}}^{f,\mathsf{poly}} Sys_2$ *(*computationally at least as secure as*) if the same as in a) holds with configurations from* $\mathsf{Conf}^f_{\mathsf{poly}}(Sys_1)$ *and* $\mathsf{Conf}_{\mathsf{poly}}(Sys_2)$ *and computational indistinguishability of the families of views.*

In all cases, we call $conf_2$ *an indistinguishable configuration for* $conf_1$*. Where the difference between the types of security is irrelevant, we simply write* $\geq_{\mathsf{sec}}^{f,}$*, and we omit the indices* f *and* sec *if they are clear from the context.* ◇

Definition 2.10 *(Blackbox and Universal Simulatability) Universal simulatability means that* A_2 *in Definition 2.9 does not depend on* H *(only on* $\hat{M}_1$*,* S_1*, and* A_1*). Blackbox simulatability means that additionally,* A_2 *(given* $\hat{M}_1$*,* S_1*) is a fixed simulator* Sim *with* A_1 *as a blackbox submachine.* ◇

2.4.2 Basic Lemmas

This section contains some lemmas that are important for working with the security definitions.

Definition 2.11 *(Combination of Machines) The* open *and* hiding combination*,* $\mathsf{D_o}$ *and* $\mathsf{D_h}$*, of a subset* $\hat{D} \subseteq \hat{C}$ *of a collection are each a machine with all the original machines as submachines. While* $Ports_{\mathsf{D_o}} := \mathsf{ports}(D)$*, in* $\mathsf{D_h}$ *internal connections are hidden, i.e.,* $Ports_{\mathsf{D_h}} := \mathsf{free}(\hat{D})$*. Both are clocked whenever a machine from* $\hat{D}$ *is. The transition function is defined by switching the submachines in the same subrounds where they would be clocked externally (and in the computational case with the same possibly shortened inputs), and* $\mathsf{D_h}$ *also buffers messages on internal connections until the receiving submachine switches next.* ◇

Lemma 2.2 *(Combination of Machines)*

a) *Let combinations as in Definition 2.11 be given. In the resulting collection* $\hat{C}^* := \hat{C} \setminus \hat{D} \cup \{\mathsf{D_x}\}$*, where* $\mathsf{x} \in \{\mathsf{o}, \mathsf{h}\}$*, the restriction of the runs to every tuple of original machines or ports is the same as in* $\hat{C}$*. (The runs as such get a slightly different representation.)*

b) *If all machines in* $\hat{D}$ *are polynomial-time, then so are* $\mathsf{D_o}$ *and* $\mathsf{D_h}$*.*

c) *Assume that only* $\mathsf{D_x}$ *is clocked in a continuous range of subrounds, i.e.,* $\kappa(i + j \bmod n) = \{\mathsf{D_x}\}$ *for* $j = 0, \ldots, l$*. Then one can replace it by a machine* D'_x *clocked only once in these subrounds, which internally calls the transition functions of its submachines in the original order. The restriction of the runs to every tuple of original machines or ports is still unchanged except for this subround renaming. We can either retain the now empty subrounds or renumber the non-empty ones.*

□

Proof. For Part a), it is clear that $\hat{C}^*$ is again a collection (and closed if $\hat{C}$ was). The rest follows immediately from the definition of the transition functions and restrictions. For b), the number of steps of the combination is the sum of the steps of the submachines, plus an overhead for internal switching linear in the length of the messages written. All this is polynomial in the length of the initial states. Part c) again follows directly from the definition. ■

Lemma 2.3 *(Indistinguishability)*

a) The statistical distance $\Delta(\phi(\mathsf{var}_k), \phi(\mathsf{var}'_k))$ between a function of two random variables is at most $\Delta(\mathsf{var}_k, \mathsf{var}'_k)$.

b) Perfect indistinguishability of two families of random variables implies perfect indistinguishability of any function ϕ of them (in particular restrictions). The same holds for statistical indistinguishability with any class SMALL, and for computational indistinguishability if ϕ is polynomial-time computable.

c) Perfect indistinguishability implies statistical indistinguishability for every non-empty class SMALL, and statistical indistinguishability for a class SMALL $\subseteq$ NEGL implies computational indistinguishability.

d) All three types of indistinguishability are equivalence relations.

□

These are well-known "folklore" facts; hence we omit the easy proof. (The proof of Part b) uses Part a), and for d) recall that the class *SMALL* must be closed under addition.)

Lemma 2.4 *(Types of Security) If $Sys_1 \geq_{\mathsf{sec}}^{f,\mathsf{perf}} Sys_2$, then also $Sys_1 \geq_{\mathsf{sec}}^{f,SMALL} Sys_2$ for every non-empty class SMALL. Similarly, $Sys_1 \geq_{\mathsf{sec}}^{f,SMALL} Sys_2$ for a class SMALL $\subseteq$ NEGL implies $Sys_1 \geq_{\mathsf{sec}}^{f,\mathsf{poly}} Sys_2$.* □

Proof. The first part follows immediately from Lemma 2.3 with the fact that equality of possibly infinite views implies equality of all their fixed-length prefixes; the second part with the fact that the view of H in a polynomial configuration is of polynomial length and that the distinguisher is a special case of a function ϕ. ■

Lemma 2.5 *(Transitivity) If $Sys_1 \geq^{f_1} Sys_2$ and $Sys_2 \geq^{f_2} Sys_3$ define $f_3 := f_2 \circ f_1$ in a natural way by $f_3(\hat{M}_1, S_1) := \bigcup_{(\hat{M}_2, S_2) \in f_1(\hat{M}_1, S_1)} f_2(\hat{M}_2, S_2)$.*

Then $Sys_1 \geq^{f_3} Sys_3$, unless f_3 is not a valid mapping. This holds for perfect, statistical and computational security, and also for universal and blackbox simulatability. □

Proof. Let a configuration $\mathit{conf}_1 = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^{f_3}(\mathit{Sys}_1)$ be given. Hence, by the definition of suitable configurations, if the name of a port p of H also occurs in some $(\hat{M}_3, S_3) \in f_3(\hat{M}_1, S_1)$, then $p^c \in S_3$.

If H has forbidden ports w.r.t. a structure $(\hat{M}_2, S_2) \in f_1(\hat{M}_1, S_1)$, we (temporarily) give these ports new names. By Lemma 2.1, we obtain a configuration $\mathit{conf}_{\mathsf{f},1} = (\hat{M}_1, S_1, \mathsf{H}_\mathsf{f}, \mathsf{A}_{\mathsf{f},1}) \in \mathsf{Conf}^{f_1}(\mathit{Sys}_1)$, where $\mathit{view}_{\mathit{conf}_{\mathsf{f},1}}(\mathsf{H}_\mathsf{f})$ equals $\mathit{view}_{\mathit{conf}_1}(\mathsf{H})$ except for the renaming.

Now there exists a configuration $\mathit{conf}_{\mathsf{f},2} = (\hat{M}_2, S_2, \mathsf{H}_\mathsf{f}, \mathsf{A}_{\mathsf{f},2}) \in \mathsf{Conf}(\mathit{Sys}_2)$ with $(\hat{M}_2, S_2) \in f_1(\hat{M}_1, S_1)$ such that $\mathit{view}_{\mathit{conf}_{\mathsf{f},1}}(\mathsf{H}_\mathsf{f}) \approx \mathit{view}_{\mathit{conf}_{\mathsf{f},2}}(\mathsf{H}_\mathsf{f})$.

As H_f only has ports from H and new ports, it has no forbidden ports w.r.t. any structure $(\hat{M}_3, S_3) \in f_2(\hat{M}_2, S_2)$, i.e., $\mathit{conf}_{\mathsf{f},2} \in \mathsf{Conf}^{f_2}(\mathit{Sys}_2)$. Hence there exists $\mathit{conf}_{\mathsf{f},3} = (\hat{M}_3, S_3, \mathsf{H}_\mathsf{f}, \mathsf{A}_{\mathsf{f},3}) \in \mathsf{Conf}(\mathit{Sys}_3)$ with $(\hat{M}_3, S_3) \in f_2(\hat{M}_2, S_2)$ and $\mathit{view}_{\mathit{conf}_{\mathsf{f},2}}(\mathsf{H}_\mathsf{f}) \approx \mathit{view}_{\mathit{conf}_{\mathsf{f},3}}(\mathsf{H}_\mathsf{f})$.

Together, we have $(\hat{M}_3, S_3) \in f_3(\hat{M}_1, S_1)$ by the definition of f_3 and $\mathit{view}_{\mathit{conf}_{\mathsf{f},1}}(\mathsf{H}_\mathsf{f}) \approx \mathit{view}_{\mathit{conf}_{\mathsf{f},3}}(\mathsf{H}_\mathsf{f})$ because indistinguishability is transitive.

Finally, we must derive a configuration $\mathit{conf}_3 = (\hat{M}_3, S_3, \mathsf{H}, \mathsf{A}_3)$ with the original user H. The new names of the changed ports do not occur in $(\hat{M}_3, S_3)$ by construction. Thus we can change them back iff the old names also do not occur in $(\hat{M}_3, S_3)$, i.e., in this case $\mathit{view}_{\mathit{conf}_3}(\mathsf{H})$ equals $\mathit{view}_{\mathit{conf}_{\mathsf{f},3}}(\mathsf{H}_\mathsf{f})$ except for this renaming. We had assured that the name of a port p of H can only occur in $(\hat{M}_3, S_3)$ if $p^c \in S_3$. It was changed if p occurred in $\mathsf{forb}(\hat{M}_2, S_2)$. As f_2 is a valid mapping, this is not possible together.

We conclude that $\mathit{view}_{\mathit{conf}_1}(\mathsf{H}) \approx \mathit{view}_{\mathit{conf}_3}(\mathsf{H})$ because the same renaming transforms these views into $\mathit{view}_{\mathit{conf}_{\mathsf{f},1}}(\mathsf{H}_\mathsf{f})$ and $\mathit{view}_{\mathit{conf}_{\mathsf{f},3}}(\mathsf{H}_\mathsf{f})$, respectively. ■

2.5 Standard Cryptographic Systems

In this section, we refine the general definitions for a standard class of systems. The intuition is that in a real system *Sys*, there is one machine per human user, and each machine is correct if and only if its user is honest. Thus it corresponds to malicious faults, and the corrupted machines remain fixed within a system run. The system is derived from an *intended structure* $(\hat{M}^*, S^*)$ and a *trust model.*

Definition 2.12 *(Standard Cryptographic Structures and Trust Model)*

a) A standard cryptographic structure *is a structure* $(\hat{M}^*, S^*)$ *where we denote the machines by* $\hat{M}^* = \{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$*. Each machine* M_u *has two ports* $\mathsf{in}_u?$ *and* $\mathsf{out}_u!$ *intended for its user, i.e., these ports are free and their union is* S^**. All other ports have complements at other machines, i.e., they correspond to a connection graph* $\mathsf{G}(\hat{M}^*)$ *among the machines from* $\hat{M}^*$*.*

b) A standard trust model *for such a structure consists of an access structure and a channel model.*

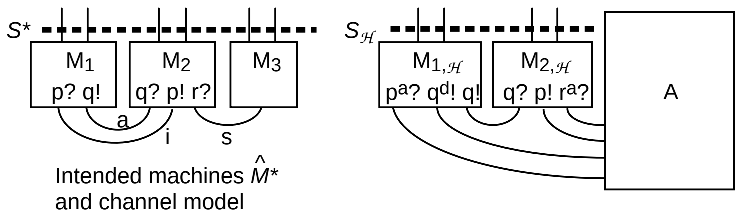

Figure 2.3: Example of a standard cryptographic system. The derived structure on the right is for $\mathcal{H} = \{1, 2\}$.

An access structure, *as usual, is a set* $\mathcal{ACC} \subseteq \mathcal{P}(\{1, \ldots, n\})$ *closed under insertion (of more elements into a set) and denotes the possible sets* $\mathcal{H}$ *of correct machines.*

A channel model *is a mapping* $\chi : \mathsf{G}(\hat{M}^*) \rightarrow \{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$. *It characterizes each connection as secure (private and authentic), authentic (only), or insecure (neither private nor authentic).*

◇

Typical examples of access structures are threshold structures $\mathcal{ACC}_t := \{\mathcal{H} \mid |\mathcal{H}| \geq t\}$ for some t, e.g., for distributed trusted third parties. In other examples like secure channels, see Chapter 3, everybody should be secure on their own, and thus $\mathcal{ACC}$ is the entire powerset $\mathcal{P}(\{1, \ldots, n\})$.

The following derivation is illustrated in Figure 2.3.

Definition 2.13 *(Standard Cryptographic Systems) Given a standard cryptographic structure* $(\hat{M}^*, S^*)$ *and a trust model* $(\mathcal{ACC}, \chi)$ *for it, the corresponding system* with static adversary *is of the form* $Sys = \{(\hat{M}_\mathcal{H}, S_\mathcal{H}) | \mathcal{H} \in \mathcal{ACC}\}$. *The specified ports are the user ports of the correct machines,*

$$S_\mathcal{H} = \{\mathsf{in}_u?, \mathsf{out}_u! | u \in \mathcal{H}\}.$$

The correct machines are

$$\hat{M}_\mathcal{H} = \{\mathsf{M}_{u,\mathcal{H}} | u \in \mathcal{H}\},$$

where each machine $\mathsf{M}_{u,\mathcal{H}}$ *is derived from* M_u *according to the channel model: Let any port* $p \in Ports_{\mathsf{M}_u} \setminus S_\mathcal{H}$ *be given, let* $p^c \in Ports_{\mathsf{M}_v}$ *and let* $c = \{p, p^c\}$ *be the attached connection.*[6]

- *If* $v \in \mathcal{H}$ *(i.e., c connects two correct machines):*
  - *If* $\chi(c) = \mathsf{s}$ *(secure), p is unchanged.*

[6]We rename more ports than necessary, for clarity in the examples: all inputs from A are on ports of type p^a.

  - *If $\chi(c) = \mathsf{a}$ (authentic) and p is an output port, $\mathsf{M}_{u,\mathcal{H}}$ gets an additional new port p^d, where it duplicates the outputs at p.*[7,8] *This port automatically remains free, and thus the adversary connects to it. If p is an input port, it is unchanged.*
  - *If $\chi(c) = \mathsf{i}$ (insecure) and p is an input port, p is replaced by a new port p^a. (Thus both p^a and p^c become free, i.e., the adversary can get the outputs from p^c and make the inputs to p^a and thus completely control the connection.) If p is an output port, it is unchanged.*

- *If $v \notin \mathcal{H}$ (i.e., there is no machine $\mathsf{M}_{v,\mathcal{H}}$ in $\hat{M}_{\mathcal{H}}$), and p is an output port, it is unchanged. If it is an input port, it is renamed into p^a. (In both cases it becomes free, and thus the adversary can connect to it.)*

◇

Definition 2.14 *(Ideal Systems and Canonical Mapping) A typical ideal system is of the form $Sys_2 = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) | \mathcal{H} \in \mathcal{ACC}\}$ with the same sets $S_{\mathcal{H}}$ as in the corresponding real systems Sys_1.*

The canonical mapping *f between such systems is defined by $f(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}})\}$ for all $\mathcal{H} \in \mathcal{ACC}$.* ◇

Remark 2.1. We cannot use the original S^* in all structures of an ideal system. Otherwise, an honest user with $Ports_{\mathsf{H}} \supseteq S^{*c}$ would be suitable for all structures $(\hat{M}_1, S_1) \in Sys_1$. It would therefore use the ports of incorrect machines in the real system, but there an adversary can behave in a different way than the trusted host would.

In cases without tolerable imperfections, i.e., if adversaries should not have any advantage over honest users, $\mathsf{TH}_{\mathcal{H}}$ would be the same in all these structures and only $S_{\mathcal{H}}$ would vary, i.e., be the same as in Sys_1. However, standard cryptographic systems will not be as secure as such a definition at least because of differences in the precise timing, see the example in Chapter 3. ∘

Remark 2.2. For large classes of cryptographic examples, one could carry the specialization further, in particular for systems with two-party transactions. Those would be more similar to concrete examples for which reactive security has been proven before, e.g., [10, 56]. One would define that each input contains the identity of a desired partner and starts a submachine (called "oracle" or "session"). However, at the level of this paper such specializations are not helpful.

Remark 2.3. It is possible to define other useful special cases. In particular, crash failures can be modeled as follows: A machine M_u is replaced by a machine $\mathsf{M}''_{\mathsf{u}}$ that acts identically, but has an additional free input port $\mathsf{stop}_{\mathsf{u}}?$ and stops if 1 is input there. ∘

[7]This and subsequent simple machine modifications can be made by canonical blackbox constructions; we tacitly assume that those are used.

[8]We assume w.l.o.g. that there is a systematic naming scheme for such new ports (e.g., appending "d") that does not clash with prior names.

2.6 Model Variants

In this section, we define variants of the model and investigate whether they are equivalent to our standard model. By "standard model", "standard user" etc., we now mean Definitions 2.5 to 2.10. The results can be summarized as follows:

- Considering an output "guess" of the adversary in addition the user view makes no difference.
- For "non-increasing" valid mappings, i.e., if $S_2 \subseteq S_1$ whenever $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$:
  - One only needs to consider users (and no adversaries) in the configurations of Sys_1.
  - Universal and blackbox definitions are equivalent.
  - A "dialogue model" with unsynchronized users is equivalent to the standard model.
- Restricting honest users to the specified ports allows more general valid mappings ("symmetrically valid"), but for the standard valid mappings it is equivalent.

The reductions relating these definitions are quite efficient. Hence our definition captures most of the versions that one might naturally consider.

Remark 2.4. We can also show that auxiliary inputs, which are important in definitions of non-reactive simulatability (e.g., zero-knowledge proofs), already give universality except in a case with very powerful honest users; in universal and blackbox definitions they make no further difference [44]. ∘

2.6.1 Guessing-Output Model

In the following guessing-output model, we also consider a final output of the adversary, or all outputs the adversary makes to some unconnected result port. This is like the guessing-outputs in semantic security [26], and essential in current definitions of multi-party function evaluation. It corresponds to the intuition that confidentiality means "whatever the *adversary* sees is simulatable," while "whatever the honest users see is simulatable" is integrity. However, we show that it makes no difference in the reactive case. Intuitively, with general active attacks, anything the adversary sees can come back to some honest users.

The definitions are modified as follows: Systems and valid mappings are unchanged. The configurations are denoted $\mathsf{Conf}_\mathsf{g}(Sys)$; the difference to $\mathsf{Conf}(Sys)$ is that for $conf_\mathsf{g} = (\hat{M}, S, \mathsf{H}, \mathsf{A}_\mathsf{g}) \in \mathsf{Conf}_\mathsf{g}(Sys)$, the adversary A_g has a reserved unconnected port $\mathsf{guess!}$ (compare Figure 2.4). "Reserved" means that the name $\mathsf{guess?}$ must not occur in the systems in this section. By $view_{conf_\mathsf{g}}(\mathsf{H}, \mathsf{guess!})$ we denote the family of restrictions of the runs to the view of H and the outputs at $\mathsf{guess!}$. "As secure as" in the guessing-output model is written "$\geq^f_{\mathsf{sec,g}}$" and defined naturally with the modified views and configurations; we spell this out in this first case for clarity:

Definition 2.15 *(Simulatability with Guessing-Output) Let two systems Sys_1 and Sys_2 with a valid mapping f be given. We call Sys_1 at least as secure as Sys_2 for f in the guessing-output model, $Sys_1 \geq^f_{\mathsf{sec,g}} Sys_2$, if for every configuration $conf_{\mathsf{g},1} = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_{\mathsf{g},1}) \in \mathsf{Conf}^f_{\mathsf{g}}(Sys_1)$, there exists $conf_{\mathsf{g},2} = (\hat{M}_2, S_2, \mathsf{H}, \mathsf{A}_{\mathsf{g},2}) \in \mathsf{Conf}_{\mathsf{g}}(Sys_2)$ with $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ such that*

$$view_{conf_{\mathsf{g},1}}(\mathsf{H}, \mathsf{guess!}) \approx view_{conf_{\mathsf{g},2}}(\mathsf{H}, \mathsf{guess!}).$$

Recall that $\approx$ means perfect, statistical, or computational indistinguishability, respectively, and for the statistical case refers to all families of polynomial-length prefixes. $\diamond$

We again omit the indices f and sec at $\geq$ if they are clear from the context. The universal and blackbox definitions are adapted in the same way.

Theorem 2.1 *(Guessing-Output) For all systems Sys_1, Sys_2 with a valid mapping f, we have*

$$Sys_1 \geq^f_{\mathsf{g}} Sys_2 \quad \Leftrightarrow \quad Sys_1 \geq^f Sys_2.$$

This is true for perfect, statistical and computational security, and also in the universal and blackbox models. $\Box$

Proof. We can treat perfect, statistical and computational security together. In the third case all given adversaries and users are polynomial-time; this will imply that so are all the constructed ones, see Lemma 2.2.

"$\geq_{\mathsf{g}} \Rightarrow \geq$": (This is the easy direction, but we do it in detail once.) Let $Sys_1 \geq_{\mathsf{g}} Sys_2$, and let $conf_1 = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^f(Sys_1)$ be given. Let $\mathsf{A}_{\mathsf{g},1}$ be A_1 augmented by a port $\mathsf{guess!}$ where it does not make any outputs. (W.l.o.g., the name guess does not occur in H and A_1 either yet.) Then $conf_{\mathsf{g},1} = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_{\mathsf{g},1}) \in \mathsf{Conf}^f_{\mathsf{g}}(Sys_1)$. Clearly, $view_{conf_{\mathsf{g},1}}(\mathsf{H}) = view_{conf_1}(\mathsf{H})$. By the precondition, there exists $conf_{\mathsf{g},2} = (\hat{M}_2, S_2, \mathsf{H}, \mathsf{A}_{\mathsf{g},2}) \in \mathsf{Conf}_{\mathsf{g}}(Sys_2)$ with $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ and $view_{conf_{\mathsf{g},1}}(\mathsf{H}, \mathsf{guess!}) \approx view_{conf_{\mathsf{g},2}}(\mathsf{H}, \mathsf{guess!})$. This implies $view_{conf_{\mathsf{g},1}}(\mathsf{H}) \approx view_{conf_{\mathsf{g},2}}(\mathsf{H})$ (Lemma 2.3). Let A_2 be $\mathsf{A}_{\mathsf{g},2}$ except that it suppresses the guessing-output. Then $conf_2 = (\hat{M}_2, S_2, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$, and clearly $view_{conf_2}(\mathsf{H}) = view_{conf_{\mathsf{g},2}}(\mathsf{H})$. Altogether, this implies $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$.

For the universal case, note that $\mathsf{A}_{\mathsf{g},1}$ is independent of H by construction, then $\mathsf{A}_{\mathsf{g},2}$ by the universal version of $\geq_{\mathsf{g}}$, and thus A_2 again by construction. In the blackbox model, we obtain $\mathsf{A}_{\mathsf{g},2}$ as a simulator Sim with $\mathsf{A}_{\mathsf{g},1}$ as a blackbox, and thus A_2 is Sim with A_1 as a blackbox.

"$\geq \Rightarrow \geq_{\mathsf{g}}$": Let $Sys_1 \geq Sys_2$, and let $conf_{\mathrm{g},1} = (\hat{M}_1, S_1, \mathsf{H}_\mathrm{g}, \mathsf{A}_{\mathrm{g},1}) \in \mathsf{Conf}^f_\mathrm{g}(Sys_1)$ be given. We construct a related configuration $conf_1 = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1)$ where the former guessing output belongs to the user's view (see Figure 2.4): The adversary A_1 equals $\mathsf{A}_{\mathrm{g},1}$, and H is like H_g except for an additional input port guess?, where it ignores all inputs. In the computational case, the explicit runtime bound of H is the maximum of those of H_g and $\mathsf{A}_{\mathrm{g},1}$.[9] We have $conf_1 \in \mathsf{Conf}^f(Sys_1)$: It is a closed collection because we required that no port guess? was there before and guess! was the only free port, and H has no forbidden ports because H_g hasn't.

Now clearly $view_{conf_1}(\mathsf{H}) = view_{conf_{\mathrm{g},1}}(\mathsf{H}_\mathrm{g}, \mathsf{guess!})$ (except that the values occur at guess? one subround later than at guess!).

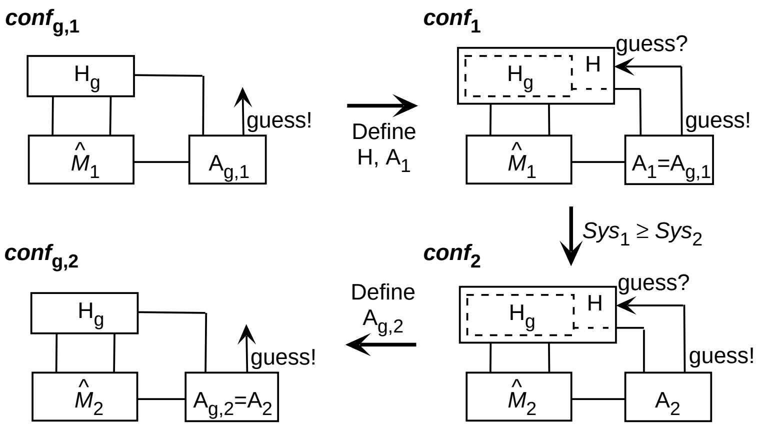

Figure 2.4: Standard simulatability implies simulatability with guessing outputs

By the precondition, there exists $conf_2 = (\hat{M}_2, S_2, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ with $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ and $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$. As assumed in the figure, A_2 must have a port guess! because H has the port guess? only, $conf_2$ is closed, and guess! cannot belong to $\hat{M}_2$ because the name is reserved.

Hence we can use $\mathsf{A}_{\mathrm{g},2} = \mathsf{A}_2$ to obtain a configuration $conf_{\mathrm{g},2} = (\hat{M}_2, S_2, \mathsf{H}_\mathrm{g}, \mathsf{A}_{\mathrm{g},2}) \in \mathsf{Conf}_\mathrm{g}(Sys_2)$. As H behaves like H_g, ignoring the inputs at guess?, the runs are essentially unchanged, and $view_{conf_{\mathrm{g},2}}(\mathsf{H}_\mathrm{g}, \mathsf{guess!}) = view_{conf_2}(\mathsf{H})$ (except the values occur at guess! one subround earlier again than at guess?).

Altogether this implies $view_{conf_{\mathrm{g},2}}(\mathsf{H}_\mathrm{g}, \mathsf{guess!}) \approx view_{conf_{\mathrm{g},1}}(\mathsf{H}_\mathrm{g}, \mathsf{guess!})$.

For the universal case, A_1 does not depend on H by construction, and hence neither do A_2 or $\mathsf{A}_{\mathrm{g},2}$. In the blackbox case, $\mathsf{A}_{\mathrm{g},2} = \mathsf{A}_2$ is a simulator with $\mathsf{A}_1 = \mathsf{A}_{\mathrm{g},1}$ as a blackbox submachine. ■

Corollary 2.1 *The equivalent of Theorem 2.1 also holds in the model where adversaries always output their entire view at the port* guess!. □

[9]This guarantees, without change to the behaviour of H_g, that the whole outputs at guess! is part of H's view.

The proof is identical to that of Theorem 2.1 except that in "$\geq_{\mathsf{g}} \Rightarrow \geq$", the adversary $\mathsf{A}_{\mathsf{g},1}$ outputs its view at the new port $\mathsf{guess!}$.

2.6.2 User-Only Model

In following user-only model, we consider users alone on the "left" side of a comparison of two systems, see Figure 2.5.

Formally, the systems are the standard ones, and we define the set $\mathsf{Conf}_{\mathsf{u}}(\mathit{Sys})$ of user-only configurations as the subset of configurations $(\hat{M}, S, \mathsf{H}, \mathsf{A}_{\mathsf{null}}) \in \mathsf{Conf}(\mathit{Sys})$, where $\mathsf{A}_{\mathsf{null}}$ is a fixed machine without ports that does nothing.[10] Furthermore, we define *non-increasing valid mappings* to be valid mappings with $[(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1) \Rightarrow S_2 \subseteq S_1]$. User-only simulatability, written $\geq^f_{\mathsf{sec,u}}$, is defined like Definition 2.9 except that conf_1 is only chosen from $\mathsf{Conf}^f_{\mathsf{u}}(\mathit{Sys}_1)$, and that only non-increasing valid mappings are allowed.

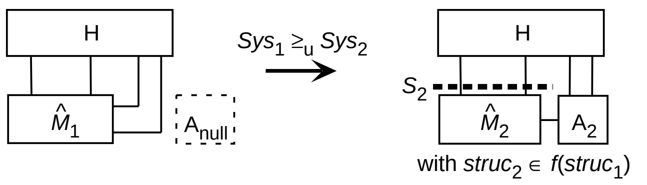
H
$\hat{M}_1$
$\mathsf{A}_{\mathsf{null}}$
$\mathit{Sys}_1 \geq_{\mathsf{u}} \mathit{Sys}_2$
H
S_2
$\hat{M}_2$
A_2
with $\mathit{struc}_2 \in f(\mathit{struc}_1)$

Figure 2.5: Definition of user-only model

Remark 2.5. The standard case for valid mappings is even $S_2 = S_1$. The reason to restrict the user-only model to non-increasing valid mappings is that we want to be able to consider users H that connect to all ports in S_2 (i.e., all ports assigned to them in the specification). For $S_2 \not\subseteq S_1$, such users cannot interact with $(\hat{M}_1, S_1)$ alone.

Remark 2.6. We have $\mathsf{Conf}^f_{\mathsf{u}}(\mathit{Sys}_1) = \mathsf{Conf}_{\mathsf{u}}(\mathit{Sys}_1)$ for every non-increasing valid mapping f: For $\mathit{conf}_{\mathsf{u},1} = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_{\mathsf{null}}) \in \mathsf{Conf}_{\mathsf{u}}(\mathit{Sys}_1)$, we have $\mathit{Ports}^c_{\mathsf{H}} = \mathsf{free}(\hat{M}_1)$. Hence by the first condition on valid mappings, no port $p \in \mathit{Ports}_{\mathsf{H}}$ can lie in $\mathsf{forb}(\hat{M}_2, S_2)$.

Remark 2.7. Universal and blackbox user-only simulatability are identical: As the adversary $\mathsf{A}_1 = \mathsf{A}_{\mathsf{null}}$ is fixed, both require that for every structure $(\hat{M}_1, S_1) \in \mathit{Sys}_1$, there is one machine A_2 such that $(\hat{M}_2, S_2, \mathsf{H}, \mathsf{A}_2)$ is an indistinguishable configuration for $(\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_{\mathsf{null}})$ for every H. ∘

Theorem 2.2 *(User-only) Let systems Sys_1 and Sys_2 with a non-increasing valid mapping f be given. Then $\mathit{Sys}_1 \geq^f_{\mathsf{u}} \mathit{Sys}_2 \Leftrightarrow \mathit{Sys}_1 \geq^f \mathit{Sys}_2$. This is true for perfect, statistical and computational security, and also for universal and blackbox definitions.* □

[10] We called the real machine present H because we treat it as a user in the clocking scheme and in simulatability. We could instead call it an adversary and regard A_2 as a blackbox simulator with relatively little power over its blackbox because many connections between the correct machines and the blackbox are unchanged.

Proof. **"$\geq \Rightarrow \geq_{\mathsf{u}}$":** This is clear in all cases because user-only simulatability is defined as a weaker version of standard simulatability.

"$\geq_{\mathsf{u}} \Rightarrow \geq$": Let a configuration $conf_1 = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1)$ be given. To transform it into a user-only configuration, we combine H and A_1 into a machine $\mathsf{H_u}$, see Figure 2.6. We define $\mathsf{H_u}$ as a hiding combination (Definition 2.11). Then only $\mathsf{H_u}$ is clocked in subrounds 2 to 4, and by Lemma 2.2 we can instead clock $\mathsf{H_u}$ once in subround 3. Thus we get the correct clocking scheme for a user machine. Hence $conf_{\mathsf{u},1} = (\hat{M}_1, S_1, \mathsf{H_u}, \mathsf{A_{null}}) \in \mathsf{Conf_u}(Sys_1) = \mathsf{Conf}^f_{\mathsf{u}}(Sys_1)$ and $view_{conf_1}(\mathsf{H})$ equals the subview of H in $view_{conf_{\mathsf{u},1}}(\mathsf{H_u})$ by Lemma 2.2.

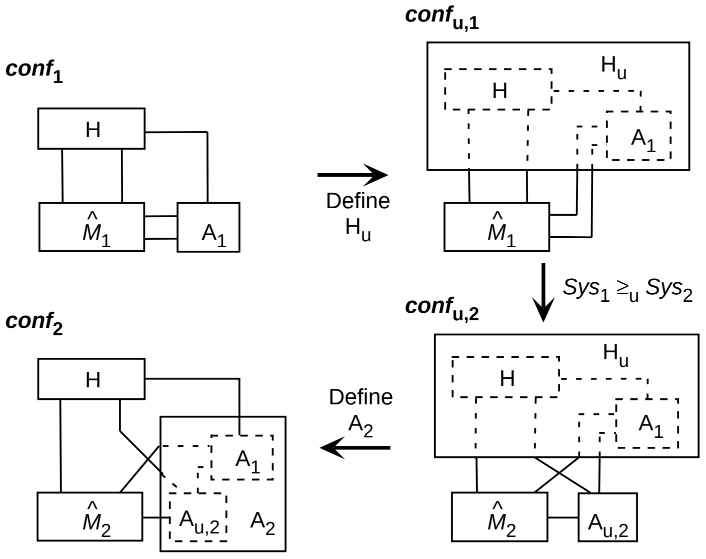

Figure 2.6: User-only simulatability implies standard simulatability

By the precondition, there exists $conf_{\mathsf{u},2} = (\hat{M}_2, S_2, \mathsf{H_u}, \mathsf{A_{u,2}}) \in \mathsf{Conf}(Sys_2)$ with $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ and $view_{conf_{\mathsf{u},1}}(\mathsf{H_u}) \approx view_{conf_{\mathsf{u},2}}(\mathsf{H_u})$.

We want to transform it into a configuration $conf_2 = (\hat{M}_2, S_2, \mathsf{H}, \mathsf{A}_2)$ with the original user H by splitting $\mathsf{H_u}$ again and then joining A_1 and $\mathsf{A_{u,2}}$. We must show that this is indeed a configuration. We show below that the ports of all machines in $\hat{M}_2$, H, A_1 and $\mathsf{A_{u,2}}$ are disjoint. Then the proof finishes as follows: We consider the intermediate closed collection $\hat{C}$ where H, A_1 and $\mathsf{A_{u,2}}$ are separate machines with the clocking scheme $(\hat{M}_2 \cup \{\mathsf{H}\}, \{\mathsf{A_{u,2}}\}, \{\mathsf{A}_1\}, \{\mathsf{H}\}, \{\mathsf{A}_1\}, \{\mathsf{A_{u,2}}\})$. Then taking the hiding combination of H and A_1, and joining subrounds 3 to 5, gives $conf_{\mathsf{u},2}$ with its correct clocking scheme, while taking the hiding combination of A_1 and $\mathsf{A_{u,2}}$, and joining subrounds 2 with 3 and 5 with 6, gives a correct configuration $conf_2 \in \mathsf{Conf}(Sys_2)$. Then, by Lemma 2.2, $view_{conf_2}(\mathsf{H})$ equals the subview of H in $view_{conf_{\mathsf{u},2}}(\mathsf{H_u})$.

Altogether, this implies $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$.

In the universal and blackbox case, $\mathsf{A}_{\mathsf{u},2}$ is independent of H_u. Thus A_2 is a fixed machine with A_1 as a blackbox submachine.

Disjoint ports. It remains to be shown that the ports of all machines in $\hat{M}_2$ and H, A_1 and $\mathsf{A}_{\mathsf{u},2}$ are disjoint.

- Clear cases: For H and $\hat{M}_2$ this holds because conf_1 is suitable (Definition 2.7), for A_1 and H because conf_1 is a collection, and for $\mathsf{A}_{\mathsf{u},2}$ and $\hat{M}_2$ because $\mathit{conf}_{\mathsf{u},2}$ is a collection.

- $\mathsf{A}_{\mathsf{u},2}$ and H: Let $p \in \mathit{Ports}_{\mathsf{A}_{\mathsf{u},2}}$. Then $p^c \in \mathit{Ports}_{\mathsf{H}_\mathsf{u}}$ or $p^c \in \mathsf{free}(\hat{M}_2)$ because $\mathit{conf}_{\mathsf{u},2}$ is closed.

  If $p^c \in \mathit{Ports}_{\mathsf{H}_\mathsf{u}}$, then $p \in \mathsf{ports}(\hat{M}_1)$ because $\mathit{conf}_{\mathsf{u},1}$ is a user-only configuration. Thus $p \notin \mathit{Ports}_\mathsf{H}$ because conf_1 is a collection. Note that then also $p \notin \mathit{Ports}_{\mathsf{A}_1}$.

  Now let $p^c \in \mathsf{free}(\hat{M}_2)$, and assume $p \in \mathit{Ports}_\mathsf{H}$. Then $p^c \notin \bar{S}_2$ because conf_1 is suitable. Thus $p^c \in S_2 \subseteq S_1 \subseteq \mathsf{ports}(\hat{M}_1)$. Hence p is connected to $\hat{M}_1$ in conf_1 and becomes a port of H_u. This contradicts the precondition $p \in \mathit{Ports}_{\mathsf{A}_{\mathsf{u},2}}$ because p cannot occur twice in $\mathit{conf}_{\mathsf{u},2}$.

- A_1 and $\hat{M}_2$: Let $p \in \mathit{Ports}_{\mathsf{A}_1}$. Then $p^c \in \mathsf{free}(\hat{M}_1)$ or $p^c \in \mathit{Ports}_\mathsf{H}$ because conf_1 is closed.

  If $p^c \in \mathsf{free}(\hat{M}_1)$, then p becomes a port of H_u. Hence $p \notin \mathsf{ports}(\hat{M}_2)$ because $\mathit{conf}_{\mathsf{u},2}$ is a collection. Note that then also $p \notin \mathit{Ports}_{\mathsf{A}_{\mathsf{u},2}}$.

  If $p^c \in \mathit{Ports}_\mathsf{H}$, then $[p \in \mathsf{ports}(\hat{M}_2) \Rightarrow p \in S_2 \subseteq S_1 \subseteq \mathsf{ports}(\hat{M}_1)]$. Then however, p would occur twice in conf_1.

- $\mathsf{A}_{\mathsf{u},2}$ and A_1: We have seen in the two previous items that $p \in \mathit{Ports}_{\mathsf{A}_{\mathsf{u},2}} \cap \mathit{Ports}_{\mathsf{A}_1}$ would imply $p^c \in \mathsf{free}(\hat{M}_2)$ and $p^c \in \mathit{Ports}_\mathsf{H}$. However, the ports of $\hat{M}_2$ and H are disjoint.

■

As we have now shown that the standard universal and blackbox definitions are both equivalent to their counterparts in the user-only model, which are equal, we have the following corollary:

Corollary 2.2 *Universal and blackbox simulatability according to Definition 2.10 are equivalent for non-increasing valid mappings.* □

2.6.3 Dialogue Model

In Section 2.3.2 we introduced our clocking scheme as a consequence of the fact that honest users need not be synchronized with the system rounds. Then, however, A and H might also engage in a multi-round dialogues within a round of the correct machines. We now show that this makes no difference, at least for non-increasing valid mappings, which includes the standard case.

In the dialogue model, systems are unchanged as always. We only allow an arbitrary but fixed number λ of subrounds, so that we still have clocking schemes according to Definition 2.2: $\kappa_\lambda(1) = \hat{M} \cup \{\mathsf{H}\}$, and then A is clocked in all even subrounds and H in all odd ones up to λ. Hence $\mathsf{Conf}_\mathsf{d}(\mathit{Sys})$ is defined as the set of pairs (conf, λ) where *conf* is a standard configuration and $\lambda \in \mathbb{N}$, and the runs of *conf* are defined with the clocking scheme κ_λ.

Dialogue simulatability, $\geq^f_{\mathsf{sec,d}}$, is defined like standard simulatability with dialogue configurations with the same λ on both sides, and for non-increasing valid mappings.

Theorem 2.3 *(Dialogues) Let systems Sys_1 and Sys_2 with a non-increasing valid mapping f be given. Then $\mathit{Sys}_1 \geq^f_\mathsf{d} \mathit{Sys}_2 \Leftrightarrow \mathit{Sys}_1 \geq^f \mathit{Sys}_2$. This is true for perfect, statistical and computational security, and also for universal and blackbox definitions.* □

Proof. We use Theorem 2.2, i.e., that all the variants of the standard model are equivalent to their user-only counterparts for non-increasing valid mappings. We treat the perfect, statistical and computational case together.

"$\geq_\mathsf{d} \Rightarrow \geq_\mathsf{u}$": Let $\mathit{conf}_1 \in \mathsf{Conf}_\mathsf{u}(\mathit{Sys}_1)$ be given. Then $(\mathit{conf}_1, 4) \in \mathsf{Conf}_\mathsf{d}(\mathit{Sys}_1)$. Thus there exists an indistinguishable configuration $(\mathit{conf}_2, 4)$ for it in $\mathsf{Conf}_\mathsf{d}(\mathit{Sys}_2)$. Hence $\mathit{conf}_2 \in \mathsf{Conf}(\mathit{Sys}_2)$ is an indistinguishable configuration for conf_1. The uniform and blackbox case are proved in the same way.

"$\geq_\mathsf{u} \Rightarrow \geq_\mathsf{d}$": One can easily see that the part "$\geq_\mathsf{u} \Rightarrow \geq$" of the proof of Theorem 2.2 also holds if a dialogue configuration $(\mathit{conf}_1, \lambda)$ is given. For machine $\mathsf{H_u}$, we join all subrounds 2 to λ. The clocking scheme κ_C of the intermediate collection $\hat{C}$ is $\kappa_C(1) = \hat{M}_2 \cup \{\mathsf{H}\}$; $\kappa_C(2) = \{\mathsf{A}_{\mathsf{u},2}\}$; $\kappa_C(i) = \kappa_\lambda(i-1)$ for $i = 3, \ldots, \lambda+1$; and $\kappa_C(\lambda+2) = \{\mathsf{A}_{\mathsf{u},2}\}$. We join subrounds 3 to $\lambda+1$ to get $\mathit{conf}_{\mathsf{u},2}$; and we join subround 2 with 3 and $\lambda+1$ with $\lambda+2$ to obtain $(\mathit{conf}_2, \lambda)$. ■

Remark 2.8. It makes no difference in this proof whether H also switches in subround $[i.1]$ or not. Hence the clocking scheme $(\hat{M}, \{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{A}\})$ is also as general as its dialogue counterpart. ○

2.6.4 Specified-User-Interface Model

In particular in cryptographic examples, we expect honest users of a structure $(\hat{M}, S)$ to use precisely the set S of specified ports and to leave the other free ports to the adversary. In the following specified-user-interface model, we consider the slightly more general set $\mathsf{Conf}_{\mathsf{s}}(\mathit{Sys}) \subseteq \mathsf{Conf}(\mathit{Sys})$ of configurations with $\mathsf{ports}(\mathsf{H}) \cap \bar{S}^c = \emptyset$. The conditions on a valid mapping are relaxed: The precondition "$p^c \in \mathsf{free}(\hat{M}_1)$" of the first condition is replaced by "$p^c \in S_1$". We call this *symmetrically valid mappings.*[11]

Theorem 2.4 *(Specified user interface) Let systems Sys_1 and Sys_2 with a valid mapping f be given. Then $\mathit{Sys}_1 \geq_{\mathsf{s}}^f \mathit{Sys}_2 \Leftrightarrow \mathit{Sys}_1 \geq^f \mathit{Sys}_2$. This is true for perfect, statistical and computational security, and also for the universal and blackbox model.* □

Proof. **"$\geq_{\mathsf{s}} \Rightarrow \geq$":** Let a configuration $\mathit{conf}_1 = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^f(\mathit{Sys}_1)$ be given. We construct a related specified-user-interface configuration $\mathit{conf}_{\mathsf{s},1} = (\hat{M}_1, S_1, \mathsf{H}_{\mathsf{s}}, \mathsf{A}_{\mathsf{s},1})$ as in Figure 2.7: H_{s} equals H except that any port p of H with $p^c \in \bar{S}_1$ gets a new name p^a. $\mathsf{A}_{\mathsf{s},1}$ equals A_1 except that it gets additional ports p and p^{ac} for all ports renamed in H_{s}, and it simply forwards messages between p and p^{ac}. Clearly $\mathit{conf}_{\mathsf{s},1}$ is still suitable and thus in $\mathsf{Conf}_{\mathsf{s}}^f(\mathit{Sys}_1)$. There is no significant difference in the executions because $\mathsf{A}_{\mathsf{s},1}$ is clocked in subrounds between H_{s} and $\hat{M}_1$. Hence $\mathit{view}_{\mathit{conf}_{\mathsf{s},1}}(\mathsf{H}_{\mathsf{s}})$ equals $\mathit{view}_{\mathit{conf}_1}(\mathsf{H})$ except for the renaming.

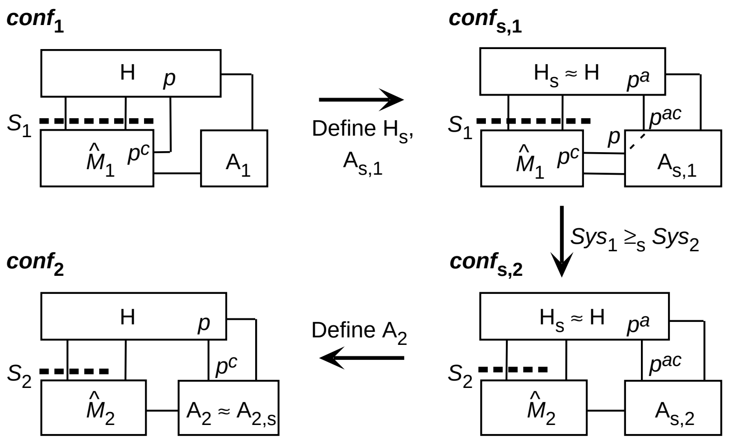

Figure 2.7: Specified-user-interface simulatability implies standard simulatability

By the precondition there exists $\mathit{conf}_{\mathsf{s},2} = (\hat{M}_2, S_2, \mathsf{H}_{\mathsf{s}}, \mathsf{A}_{\mathsf{s},2}) \in \mathsf{Conf}_{\mathsf{s}}(\mathit{Sys}_2)$ with $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ and $\mathit{view}_{\mathit{conf}_{\mathsf{s},1}}(\mathsf{H}_{\mathsf{s}}) \approx \mathit{view}_{\mathit{conf}_{\mathsf{s},2}}(\mathsf{H}_{\mathsf{s}})$. As assumed in the figure, $\mathsf{A}_{\mathsf{s},2}$ has a port p^{ac} for every new port p^a of H_{s} because the collection is closed and p^{ac} cannot belong to H_{s} or $\hat{M}_2$ because the name was new.

[11]Lemma 2.1 and the proof of transitivity can easily be adapted to this case. Hence symmetrically valid mappings at the cost of more restricted users are a true alternative to our standard definition.

We want to transform this into a configuration $conf_2 = (\hat{M}_2, S_2, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ with the original H by giving each renamed port p^a of $\mathsf{H_s}$ its old name p. To retain the connection, we also rename p^{ac} of $\mathsf{A}_{\mathsf{s},2}$ into p^c in A_2. We have to make sure that p, p^c do not occur elsewhere in $conf_{\mathsf{s},2}$. By the choice of p, we have $p^c \notin Ports_{\mathsf{H}}$. As $conf_1$ is suitable, $p \notin \mathsf{ports}(\hat{M}_2)$ and $[p^c \in \mathsf{ports}(\hat{M}_2) \Rightarrow p^c \in S_2]$. The latter and the validity of f would imply $p^c \notin \bar{S}_1$ in contradiction to our choice of p. Hence p and p^c can only occur in $\mathsf{A}_{\mathsf{s},2}$. Then we also give these ports new names in A_2. We have therefore ensured $view_{conf_2}(\mathsf{H})$ $= view_{conf_{\mathsf{s},2}}(\mathsf{H_s})$ except for the renaming.

Altogether, this implies $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$.

"$\geq \Rightarrow \geq_{\mathsf{s}}$": Let $conf_{\mathsf{s},1} = (\hat{M}_1, S_1, \mathsf{H_s}, \mathsf{A}_{\mathsf{s},1}) \in \mathsf{Conf}_{\mathsf{s}}^f(Sys_1) \subseteq \mathsf{Conf}^f(Sys_1)$ be given. Hence there exists $conf_2 = (\hat{M}_2, S_2, \mathsf{H_s}, \mathsf{A}_2)$ with $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ and $view_{conf_{\mathsf{s},1}}(\mathsf{H_s}) \approx view_{conf_2}(\mathsf{H_s})$. Suitability of $conf_{\mathsf{s},1}$ implies $\mathsf{ports}(\mathsf{H_s}) \cap \bar{S}_2^c = \emptyset$, i.e., $conf_2 \in \mathsf{Conf_s}(Sys_2)$. ■

For non-increasing valid mappings, we can restrict ourselves to user-only configurations $conf_1$ in the part "$\geq_{\mathsf{s}} \Rightarrow \geq$" of this proof. Then $\mathsf{H_s}$ uses all ports from S_1. Hence we obtain the following corollary:

Corollary 2.3 *If f is non-increasing, the same theorem holds for users that use* precisely *the specified ports, i.e., configurations with* $\mathsf{ports}(\mathsf{H})^c \cap \mathsf{free}(\hat{M}) = S$. □

2.7 Integrity Requirements

In this section, we show how the relation "at least as secure as" relates to explicit properties required of a system, e.g., safety requirements expressed in temporal logic.

In a modular design approach, one regards the trusted host, i.e., the ideal system used as the specification in simulatability, as a refinement of these properties. Hence one verifies these properties for the trusted host. This may be done by formal and even automatic model checking if the trusted host is simple enough. (The trusted hosts in our two larger examples in Chapter 3 and [45] are indeed without probabilistic and computational aspects or cryptographic operations.) Now we want to show that the real system also fulfills these requirements in a certain cryptographic sense, i.e., even if parts of the system are under control of an adversary, but possibly only for polynomial-time adversaries and with small error probabilities. This approach corresponds to the right half of Figure 1.1: the integrity properties serve as abstract goals, the ideal system as an abstract protocol, and the real system fulfils generally defined concrete versions of these goals.

Clearly this can only hold for requirements formulated in terms of in- and outputs of the trusted host at the specified ports, because the simulatability definition only means that the real and the ideal system interact with their users in an indistinguishable way.

As a rather general version of integrity requirements, independent of the concrete formal language, we consider those that have a linear-time semantics, i.e., that correspond to a set of allowed traces of in- and outputs. We allow different requirements for different sets S of specified ports, because the requirements of various parties in cryptographic protocols are often made for different trust assumptions (typically, every party is assumed to trust only their own computer). To make the translation between the two systems meaningful, we only consider mappings f that keep S constant, i.e., $S_1 = S_2$ whenever $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$. We call this a *fixed-S valid mapping.*

Definition 2.16 *(Integrity Requirements) An integrity requirement Req for a system Sys is a function that assigns a set Req(S) of traces at the ports in S to each set S with* $(\hat{M}, S) \in \mathit{Sys}$*. Such a trace is a function* $t : S \times \mathbb{N} \to \Sigma^* \cup \{\bot\}$*, where* $t[p, i] \in \Sigma^*$ *corresponds to the in- or output at port p in Subround* $[i.1]$*. (Necessarily,* $p \in \mathit{Ports}_{\mathsf{M}}$ *for a machine* $\mathsf{M} \in \hat{M}$*.) We say that Sys fulfills Req*

a) perfectly (written $\mathit{Sys} \models_{\mathsf{perf}} \mathit{Req}$*) if for every configuration* $\mathit{conf} := (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(\mathit{Sys})$*, the restrictions* $r\lceil_S$ *of all runs of this configuration to the specified ports lie in Req(S). In formulas,* $[(\mathit{run}_{\mathit{conf},k}\lceil_S)] \subseteq \mathit{Req}(S)$ *for all k, where* $[\cdot]$ *denotes the carrier set of a probability distribution.*

b) statistically for a class SMALL $(\mathit{Sys} \models_{\mathit{SMALL}} \mathit{Req})$ *if for every configuration* $\mathit{conf} := (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(\mathit{Sys})$*, the probability that Req(S) is not fulfilled is small, i.e., for all polynomials l (and as a function of k),*

$$P(\mathit{run}_{\mathit{conf},k,l(k)}\lceil_S \notin \mathit{Req}(S)) \in \mathit{SMALL}.$$

c) computationally $(\mathit{Sys} \models_{\mathsf{poly}} \mathit{Req})$ *if for every configuration* $\mathit{conf} := (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}_{\mathsf{poly}}(\mathit{Sys})$*, the probability that Req(S) is not fulfilled is negligible, i.e.,*

$$P(\mathit{run}_{\mathit{conf},k}\lceil_S \notin \mathit{Req}(S)) \in \mathit{NEGL}.$$

Note that a) is normal fulfillment. We write "$\models$*" if we want to treat all three cases together.*
◇

Theorem 2.5 *(Integrity Properties) Let systems* Sys_1 *and* Sys_2 *with a fixed-S valid mapping f and an integrity requirement Req be given. Then*

$$\mathit{Sys}_2 \models \mathit{Req} \quad \Rightarrow \quad \mathit{Sys}_1 \models \mathit{Req}.$$

This holds in the perfect and statistical sense, and in the computational sense if the systems are polynomial-time and membership of finite traces in the set Req(S) is decidable in polynomial time for every S. □

Proof. Let $\mathit{Sys}_2 \models \mathit{Req}$. We first show that *Req* is defined on Sys_1 under the preconditions: For every $(\hat{M}_1, S) \in \mathit{Sys}_1$, there exists $(\hat{M}_2, S) \in f(\hat{M}_1, S)$, and thus $\mathit{Req}(S)$ is defined. The idea for the rest of the proof is that if Sys_1 did not fulfill the requirement while Sys_2 does, this would offer a possibility to distinguish the systems.

We use Theorem 2.2, i.e., that all the variants of the standard model are equivalent to their user-only counterparts for non-increasing valid mappings. (This helps making the traces part of the indistinguishable views).

Assume that a configuration $conf_1 := (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_{\mathsf{null}}) \in \mathsf{Conf}_{\mathsf{u}}(Sys_1)$ contradicts the theorem. For the computational case, we assume w.l.o.g. that the explicit runtime bound of H is larger than that of $\hat{M}_1$ (considered as a closed combination) and those of the corresponding $\hat{M}_2$'s. There exists an indistinguishable configuration $conf_2 := (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$, i.e., $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$. By the precondition, the requirement is fulfilled for this configuration (perfectly, statistically, or computationally). Furthermore, the trace at S is a function of the view of H in both configurations (because the view contains the trace at S^c, and the runtime of H is large enough that nothing is shortened in transport).

In the perfect case, the distribution of the views is identical. This immediately contradicts the assumption that $[(run_{conf_1,k}\lceil_S)] \not\subseteq Req(S)$ while $[(run_{conf_2,k}\lceil_S)] \subseteq Req(S)$.

In the statistical case, let an arbitrary polynomial l be given. The statistical distance $\Delta(view_{conf_1,k,l(k)}(\mathsf{H}), view_{conf_2,k,l(k)}(\mathsf{H}))$ is a function $g(k) \in SMALL$. We apply Lemma 2.3 to the characteristic function $1_{v\lceil_S \notin Req(S)}$ on such views v. This gives $|P(run_{conf_1,k,l(k)}\lceil_S \notin Req(S)) - P(run_{conf_2,k,l(k)}\lceil_S \notin Req(S))| \leq g(k)$. As $SMALL$ is closed under addition and under making functions smaller, this gives the desired contradiction.

In the computational case, we define a distinguisher Dis: Given a view of machine H, it extracts the trace at S and verifies if it lies in $Req(S)$. If yes, it outputs 0, otherwise 1. This distinguisher is polynomial-time (in the security parameter k) because the view of H is of polynomial length, and membership in $Req(S)$ was required to be polynomial-time decidable. Its advantage in distinguishing is $|P(\mathsf{Dis}(1^k, view_{conf_1,k}) = 1) - P(\mathsf{Dis}(1^k, view_{conf_2,k}) = 1)| = |P(run_{conf_1,k}\lceil_S \notin Req(S)) - P(run_{conf_2,k}\lceil_S \notin Req(S))|$. If this difference were negligible, then so would the first term be because the second term is and $NEGL$ is closed under addition. Again this is the desired contradiction. ■

We now show that, if integrity requirements are formulated in a logic (e.g., temporal logic, or first-order logic with round numbers), abstract derivations in the logic are valid in the cryptographic sense. As our definitions are not based on a specific logic, but on the linear-time semantics, the following theorem represents this (see below).

Theorem 2.6 *(Modus ponens)*

a) If $Sys \models Req_1$ and $Req_1 \subseteq Req_2$, then also $Sys \models Req_2$.

b) If $Sys \models Req_1$ and $Sys \models Req_2$, then also $Sys \models Req_1 \cap Req_2$.

Here "$\subseteq$" and "$\cap$" are interpreted pointwise, i.e., for each S. This holds in the perfect and statistical sense, and in the computational sense if for a) membership in $Req_2(S)$ is decidable in polynomial time for every S. □

Proof. Part a) is trivially fulfilled in all three cases. Part b) is trivial in the perfect case. For the statistical case and every $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(Sys)$,

$$\begin{aligned} & P(run_{conf,k,l(k)}\lceil_S \notin (Req_1(S) \cap Req_2(S)) \\ \leq\ & P(run_{conf,k,l(k)}\lceil_S \notin Req_1(S)) + P(run_{conf,k,l(k)}\lceil_S \notin Req_2(S)) \\ \in\ & SMALL \end{aligned}$$

because both summands are in $SMALL$, which is closed under addition. The computational case holds analogously because $NEGL$ is closed under addition. ■

In applying this theorem to concrete logics, the main rule to consider is usually modus ponens, i.e., if one has derived that Formulas a and $a \rightarrow b$ are valid in a given model, then b is also valid in this model. If Req_a etc. denote the semantics of the formulas, i.e., the trace sets they represent, we have to show that

$$(Sys \models Req_a \wedge Sys \models Req_{a \rightarrow b}) \ \Rightarrow\ Sys \models Req_b.$$

This follows directly from the theorem with $Req_a \cap Req_{a \rightarrow b} = Req_{a \wedge b} \subseteq Req_b$.

2.8 Composition

In this section, we show that the relation "at least as secure as" is consistent with the composition of systems. The basic idea is the following: Assume that we have proven that a system Sys_0 is as secure as another system Sys'_0 (typically an ideal system used as a specification). Now we would like to use Sys_0 as a secure replacement for Sys'_0, i.e., as an implementation of the specification Sys'_0.

Usually, replacing Sys'_0 means that we have another system Sys_1 that uses Sys'_0; we call this composition Sys^*. Inside Sys^* we want to use Sys_0 instead, which gives a composition $Sys^\#$. Hence $Sys^\#$ is typically a completely real system, while Sys^* is partly ideal. Intuitively we expect $Sys^\#$ to be at least as secure as Sys^*. The situation is shown in the left and middle part of Figure 2.8.

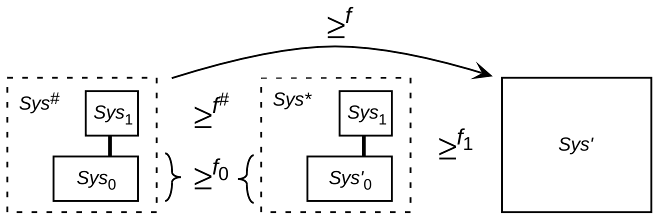

Figure 2.8: Composition theorem and its use in a modular proof: The left and middle part show the statement of Theorem 2.7, the right part Corollary 2.4.

In terms of Figure 1.1, once we have proven Sys_0, it serves as a concrete primitive and Sys'_0 as the abstract primitive. The abstract protocol is Sys_1 or, considered together with the abstract primitive, Sys^*. The concrete protocol is again Sys_1 or, together with the concrete primitive, $Sys^{\#}$. In Theorem 2.7, we show that the arrow "abstraction" in the middle of Figure 1.1 is correct. (Here "$\geq_{\mathsf{sec}}$" is the concrete version of "abstraction".) Thus we can continue the modular design with Sys^* as an abstract primitive for larger systems.

Corollary 2.4 adds the right part of Figure 2.8. It also corresponds to the right part of Figure 1.1 for the case where the abstract goals are given by a specification Sys', i.e., "fulfils" is then also "$\geq_{\mathsf{sec}}$", and the abstract and concrete goals are the same.

We first have to define composition. We do it immediately for n systems $Sys_1, \ldots, Sys_n$. We do not provide a composition operator that takes the individual systems and produces one specific composition. The reason is that one typically does not want to compose every structure of one system with every structure of the others, but only with certain matching ones. E.g., if the individual machines of $\hat{M}_1$ are implemented on the same physical devices as those of $\hat{M}_0$, as usual with a layered distributed system, we might only want to compose structures corresponding to the same trust model. However, this is not the only conceivable situation. Hence we allow many different compositions.

Definition 2.17 *(Composition) The composition of structures and of systems is defined as follows:*

1. *We call structures $(\hat{M}_1, S_1), \ldots, (\hat{M}_n, S_n)$* composable *if $\mathsf{ports}(\hat{M}_i) \cap \mathsf{ports}(\hat{M}_j) = \emptyset$ for all $i \neq j$. We then define their composition as*
$$(\hat{M}_1, S_1)|| \ldots ||(\hat{M}_n, S_n) := (\hat{M}, S)$$
with $\hat{M} := \hat{M}_1 \cup \ldots \cup \hat{M}_n$ and $S := (S_1 \cup \ldots \cup S_n) \cap \mathsf{free}(\hat{M})$. Clearly, $(\hat{M}, S)$ is again a structure.

2. *We call a system Sys a composition of systems $Sys_1, \ldots, Sys_n$ and write*
$$Sys \in Sys_1 \times \cdots \times Sys_n$$
if each structure $(\hat{M}, S) \in Sys$ has a unique representation $(\hat{M}, S) = (\hat{M}_1, S_1)|| \ldots ||(\hat{M}_n, S_n)$ with composable structures $(\hat{M}_i, S_i) \in Sys_i$ for $i = 1, \ldots, n$.

3. *Under the conditions of 2., we call $(\hat{M}_i, S_i)$ the restriction of $(\hat{M}, S)$ to Sys_i and write $(\hat{M}_i, S_i) =: (\hat{M}, S)\lceil_{Sys_i}$.*

◇

Remark 2.9. As compositions are again systems and structures, all further definitions (configurations, runs etc.) apply to them. ○

Remark 2.10. Restriction is defined relative to all n systems, i.e., uniqueness is only guaranteed if one knows them all. In most cases, however, it is unique even if one only knows $(\hat{M}, S)$ and Sys_i, e.g., if the n systems have disjoint sets of machines and each set $\hat{M}_i$ occurs in at most one structure of Sys_i. ○

The following theorem shows that modular proofs as sketched in the introduction to this section are indeed possible. Recall that the situation is shown in the left and middle part of Figure 2.8. The main issue in formulating the theorem is to characterize $\mathit{Sys}^{\#}$, i.e., to formulate what it means that Sys_0 replaces Sys'_0.

Theorem 2.7 *(Secure Two-System Composition) Let systems Sys_0, Sys'_0, Sys_1 and a valid mapping f_0 be given with*

$$\mathit{Sys}_0 \geq^{f_0} \mathit{Sys}'_0.$$

Let compositions $\mathit{Sys}^{\#} \in \mathit{Sys}_0 \times \mathit{Sys}_1$ and $\mathit{Sys}^* \in \mathit{Sys}'_0 \times \mathit{Sys}_1$ be given that fulfil the following structural conditions:

1. *For every structure $(\hat{M}^{\#}, S^{\#}) \in \mathit{Sys}^{\#}$ with restrictions $(\hat{M}_i, S_i) := (\hat{M}^{\#}, S^{\#})\lceil_{\mathit{Sys}_i}$ and every $(\hat{M}'_0, S'_0) \in f_0(\hat{M}_0, S_0)$, the composition $(\hat{M}'_0, S'_0)||(\hat{M}_1, S_1)$ exists and lies in Sys^*.*

   Let $f^{\#}$ be the function that maps each $(\hat{M}^{\#}, S^{\#})$ to the set of these compositions.

2. *If $(\hat{M}_1, S_1) \in \mathit{Sys}_1$ then $\mathsf{ports}(\hat{M}_1) \cap \mathsf{forb}(\hat{M}'_0, S'_0) = \emptyset$.*

Then we have

$$\mathit{Sys}^{\#} \geq^{f^{\#}} \mathit{Sys}^*.$$

This holds for perfect and statistical and, if Sys_1 is polynomial-time, for computational security, and also for the universal and blackbox definitions. □

Remark 2.11. Condition (2) implies that the machines in $\hat{M}_1$ can be considered part of the user for every structure $(\hat{M}'_0, S'_0)$. We could weaken the condition by only comparing structures $(\hat{M}_1, S_1) := (\hat{M}^{\#}, S^{\#})\lceil_{\mathit{Sys}_1}$ with structures $(\hat{M}'_0, S'_0) \in f_0((\hat{M}^{\#}, S^{\#})\lceil_{\mathit{Sys}_0})$. The simpler but stronger condition is intuitively w.l.o.g.: The only ports that really need to have the same names in different systems are the specified ones. All others can be given a specific prefix for each system. ○

Proof. (Of Theorem 2.7.) Let a configuration $\mathit{conf}^{\#} := (\hat{M}^{\#}, S^{\#}, \mathsf{H}, \mathsf{A}^{\#}) \in \mathsf{Conf}^{f^{\#}}(\mathit{Sys}^{\#})$ be given and $(\hat{M}_i, S_i) := (\hat{M}^{\#}, S^{\#})\lceil_{\mathit{Sys}_i}$ for $i = 0, 1$. We have to show that there is an indistinguishable configuration $\mathit{conf}^* \in \mathsf{Conf}(\mathit{Sys}^*)$. The outline of the proof is as follows; it is illustrated in Figure 2.9.

1. We combine H and $\hat{M}_1$ into a user H_0 to obtain a configuration $\mathit{conf}_0 := (\hat{M}_0, S_0, \mathsf{H}, \mathsf{A}_0) \in \mathsf{Conf}(\mathit{Sys}_0)$ where the view of H as a submachine of H_0 is the same as that in $\mathit{conf}^{\#}$.

2. We show that $\mathit{conf}_0 \in \mathsf{Conf}^{f_0}(\mathit{Sys}_0)$. Then by the precondition $\mathit{Sys}_0 \geq^{f_0} \mathit{Sys}'_0$, there is a configuration $\mathit{conf}'_0 := (\hat{M}'_0, S'_0, \mathsf{H}_0, \mathsf{A}'_0) \in \mathsf{Conf}(\mathit{Sys}'_0)$ with $(\hat{M}'_0, S'_0) \in f_0(\hat{M}_0, S_0)$ where the view of H_0 is indistinguishable from that in conf_0.

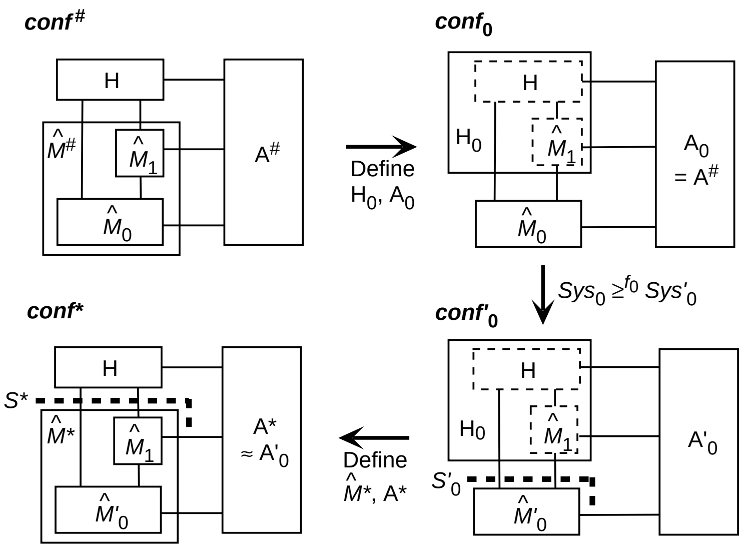

Figure 2.9: Configurations in the composition theorem. Dashed machines are internal submachines. (The connections drawn inside H_0 are not dashed because the combination is open.)

3. We decompose H_0 into H and $\hat{M}_1$ again and derive a configuration $conf^* := (\hat{M}^*, S^*, \mathsf{H}, \mathsf{A}^*) \in \mathsf{Conf}(Sys^*)$ where the view of H equals that of H as a submachine of H_0 in $conf'_0$.

4. We conclude that $conf^*$ is an indistinguishable configuration for $conf^{\#}$.

We now present the four steps in detail.

Step 1: The precise definition of $conf_0 := (\hat{M}_0, S_0, \mathsf{H}, \mathsf{A}_0)$ is that $(\hat{M}_0, S_0) := (\hat{M}^{\#}, S^{\#})\lceil_{Sys_0}$, that H_0 is the open combination of $\hat{M}_1 \cup \{\mathsf{H}\}$ as in Lemma 2.2, and $\mathsf{A}_0 := \mathsf{A}^{\#}$. This is a correct configuration from $\mathsf{Conf}(Sys_0)$:

- $(\hat{M}_0, S_0) = (\hat{M}^{\#}, S^{\#})\lceil_{Sys_0}$ is a correct structure by the definition of a composition.
- Closed collection: The overall set of ports is the same as in $conf^{\#}$. Hence the machines still have pairwise disjoint port sets, and all ports still have complements in the set.
- For the computational case, H and all machines in $\hat{M}_1$ are polynomial-time by the preconditions. Hence H_0 is polynomial-time by Lemma 2.2.

Hence Lemma 2.2 implies $view_{conf_0}(\mathsf{H}) = view_{conf^{\#}}(\mathsf{H})$.

Step 2: We now show that $conf_0 \in \mathsf{Conf}^{f_0}(Sys_0)$, i.e., H_0 has no ports from $\mathsf{forb}(\hat{M}'_0, S'_0) = \mathsf{ports}(\hat{M}'_0) \cup \bar{S}'^c_0$ for any structure $(\hat{M}'_0, S'_0) \in f_0(\hat{M}_0, S_0)$.

Assume that it had such a port p. By construction of H_0, p is also a port of $\hat{M}_1$ or H. The first case is excluded in Precondition 2 of the theorem. Thus $p \in \mathit{Ports}_{\mathsf{H}}$. We use that $\mathit{conf}^{\#}$ is suitable, i.e., H has no ports from $\mathsf{forb}(\hat{M}^*, S^*) = \mathsf{ports}(\hat{M}^*) \cup \bar{S}^{*c}$ for any $(\hat{M}^*, S^*) \in f^{\#}(\hat{M}^{\#}, S^{\#})$. By Precondition (1) of the Theorem, $(\hat{M}_0', S_0')||(\hat{M}_1, S_1)$ exists and lies in $f^{\#}(\hat{M}^{\#}, S^{\#})$, hence we use this as $(\hat{M}^*, S^*)$.

Then $\hat{M}_0' \subseteq \hat{M}^*$ implies $p \notin \mathsf{ports}(\hat{M}_0')$. Thus only $p^c \in \bar{S}_0'$ remains, and we want to show that it contradicts $p^c \notin \bar{S}^*$. We first show $p^c \in \mathsf{free}(\hat{M}^*)$: We have $p^c \in \bar{S}_0' \subseteq \mathsf{free}(\hat{M}_0')$, and thus $p \notin \mathsf{ports}(\hat{M}_0')$, and we have shown above that $p \notin \mathsf{ports}(\hat{M}_1)$. Secondly, disjointness of the port names of $\hat{M}_0'$ and $\hat{M}_1$ implies $p^c \notin S_1$. Hence $p^c \in \mathsf{free}(\hat{M}^*) \setminus (S_0' \cup S_1) = \bar{S}^*$. This is the desired contradiction.

Hence conf_0 is indeed a suitable configuration. Thus $\mathit{Sys}_0 \geq^{f_0} \mathit{Sys}_0'$ implies that there is a configuration $\mathit{conf}_0' := (\hat{M}_0', S_0', \mathsf{H}_0, \mathsf{A}_0') \in \mathsf{Conf}(\mathit{Sys}_0')$ with $(\hat{M}_0', S_0') \in f_0(\hat{M}_0, S_0)$ and $\mathit{view}_{\mathit{conf}_0'}(\mathsf{H}_0) \approx \mathit{view}_{\mathit{conf}_0}(\mathsf{H}_0)$. This implies $\mathit{view}_{\mathit{conf}_0'}(\mathsf{H}) \approx \mathit{view}_{\mathit{conf}_0}(\mathsf{H})$ because the view of a submachine is a function of the larger view (Lemmas 2.3 and 2.2).

Step 3: We define $\mathit{conf}^* := (\hat{M}^*, S^*, \mathsf{H}, \mathsf{A}^*)$ by reversing the combination of H and $\hat{M}_1$ into H_0: The structure is $(\hat{M}^*, S^*) := (\hat{M}_0', S_0')||(\hat{M}_1, S_1)$, the user the original H, and $\mathsf{A}^* := \mathsf{A}_0'$. We show that $\mathit{conf}^* \in \mathsf{Conf}(\mathit{Sys}^*)$.

- Structure: $(\hat{M}^*, S^*) \in \mathit{Sys}^*$ follows immediately from Precondition 1 of the theorem.
- Closed collection: The ports of H and the machines in $\hat{M}_1$ are disjoint because so they were in $\mathit{conf}^{\#}$, and those of all other pairs of machines because so they were in conf_0'.[12] As the set of ports is the same as in conf_0', they also still all have complements in the set.

We can now see conf_0' as derived from conf^* by taking the open combination of $\hat{M}_1 \cup \{\mathsf{H}\}$. Hence Lemma 2.2 applies, and we obtain $\mathit{view}_{\mathit{conf}^*}(\mathsf{H}) = \mathit{view}_{\mathit{conf}_0'}(\mathsf{H})$.

Step 4: We have shown that $\mathit{conf}^* \in \mathsf{Conf}(\mathit{Sys}^*)$. We also have $(\hat{M}^*, S^*) \in f^{\#}(\hat{M}^{\#}, S^{\#})$ by the construction of $f^{\#}$. The results about views in Steps 1 to 3 and transitivity (Lemma 2.5) imply that $\mathit{view}_{\mathit{conf}^*}(\mathsf{H}) \approx \mathit{view}_{\mathit{conf}^{\#}}(\mathsf{H})$. Hence conf^* is indeed an indistinguishable configuration for $\mathit{conf}^{\#}$.

Universal and blackbox: For the universal case, note that $\mathsf{A}_0 = \mathsf{A}^{\#}$ does not depend on H. Then A_0' only depends on $(\hat{M}_0, S_0)$ and A_0, and thus also $\mathsf{A}^* = \mathsf{A}_0'$. For the blackbox case, A_0' additionally consists of a simulator Sim with $\mathsf{A}_0 = \mathsf{A}^{\#}$ as a blackbox, and thus so does A^*. ■

The following corollary finishes the formalization and proof of the situation shown in Figure 2.8: We now assume that there is also a specification Sys' for the system Sys^*, as shown in the left part of the figure.

[12] Recall that $\mathit{Ports}_{\mathsf{H}_0} = \mathsf{ports}(\hat{M}_1 \cup \{\mathsf{H}\})$; here we exploit that we did not hide internal connections in the combination.

Corollary 2.4 *Consider five systems satisfying the preconditions of Theorem 2.7, and a sixth one,* Sys'*, with* $\mathit{Sys}^* \geq^{f_1} \mathit{Sys}'$*. Then*

$$\mathit{Sys}^{\#} \geq^{f} \mathit{Sys}'$$

where $f := f_1 \circ f^{\#}$ *as defined in Lemma 2.5 (transitivity), except if* f *is not a valid mapping.* □

Proof. Theorem 2.7 implies that $\mathit{Sys}^{\#} \geq^{f^{\#}} \mathit{Sys}^*$. Then we immediately obtain $\mathit{Sys}^{\#} \geq^{f}$ Sys' using transitivity (Lemma 2.5). ■

The restriction that f must be a valid mapping is not serious (as for transitivity generally); it means that the naming conventions must be fulfilled for the two systems $\mathit{Sys}^{\#}$ and Sys' that we finally want to compare.

Remark 2.12. An alternative to the corollary would be to consider *two* specifications Sys_0' and Sys_1' for the two real systems Sys_0 and Sys_1, and to show that $\mathit{Sys}^{\#} \in \mathit{Sys}_0 \times \mathit{Sys}_1$ is as secure as some $\mathit{Sys}' \in \mathit{Sys}_0' \times \mathit{Sys}_1'$. For this, our composition theorem could be applied to internally structured systems Sys'. However, typically a specification should not prescribe that the implementation must have two subsystems; e.g., in specifying a payment system it should be irrelevant whether the implementation uses secret channels as a subsystem. Hence the overall specification Sys' will typically be monolithic as in Figure 2.8. ○

3 Example of Faithful Abstraction from Cryptography

We now define and prove one reactive cryptographic system in detail, a system for secure message transmission. By secure message transmission, we mean the sending and receiving of private and authentic messages, as usually achieved by encryption and authentication. In other words, it is the provision of secure point-to-point channels, i.e., the lowest layer of the MAFTIA middleware.

The main purpose of this example is the validation of the rigorous model by an example that seems simple. Nevertheless we were not sure in advance what message format would be right, and certain timing problems and other imperfections like traffic analysis had to be dealt with. The proof is rigorous in both the cryptographic and the non-cryptographic aspects. In more detail, this example shows

- why we model tolerable imperfections, and thus allow ideal systems with different trusted hosts containing specific capabilities for adversaries,
- the timing problems that occur in a synchronous system and why we think they should not always be abstracted from,
- and that some system is actually provable with respect to our definition.

Remark 3.1. A security proof for a larger cryptographic protocol, certified mail, in the same model can be found in [45]. This type of protocol had not previously been defined rigorously or considered under a simulatability approach. It is also more similar to contract signing, the example addressed in Chapter 4 in CSP. However, the secure-message example is much more suitable for a non-specialist reader. ◦

3.1 Tolerable Imperfections and Timing Problems

Traffic Patterns and Traffic Suppression The most intuitive version of an ideal system for secure message transmission would be a trusted host that simply relays messages. E.g., [24], Page 8, says that an encryption scheme is considered secure if it simulates an ideal private channel between the parties, and that this means that an eavesdropping adversary gains nothing over a user which does not tap this channel. This is indeed the most desirable service. However, it implies that an adversary does not even see whether a message is sent or not. Thus any real system as secure as this trusted host must also hide this fact, e.g., by sending meaningless ciphertexts at all other times (see [6] for an early note of this and [1] for one in a similar context as here). This may indeed be the method of choice in a few applications, but it costs a lot of bandwidth. Usually one is therefore satisfied with encrypting the real messages only. Nevertheless, one may want to have a precise definition of what one has achieved in the form of the simulation of a trusted host.

For this, we introduce an abstraction of the tolerable imperfection into the trusted host. For encryption, the trusted host will tell the adversaries the traffic pattern in the form of one "busy" bit per pair of honest participants and round.

Similarly, as cryptography cannot prevent an adversary from destroying a message in transit, the trusted host will allow the adversary to input a bit "suppress" for any message that was signaled to him by "busy". Again, in some applications one will want a more ideal service, and thus need a lot of replication in the underlying protocol, but typically one is satisfied with the much cheaper service, and hence this should also be defined and proved. Furthermore, for availability one would need stronger assumptions on the number of corrupted machines than for privacy and integrity, where only the two parties concerned must be honest.

Observable traffic patterns also occur in many other systems, e.g., all typical key exchange and authentication protocols or payment systems. Our approach enables us to separate vulnerabilities of insecure implementations (such as cryptographic problems or protocol vulnerabilities allowing replay attacks etc.) from such unavoidable imperfections of the service given certain resource limitations.

Timing Imperfections While, once noticed, the traffic analysis imperfections may seem clear, it is less obvious that even after these changes to the initially desired trusted host, no real implementation will be as secure as it. (This would also be true if we had not allowed the imperfections discussed so far.) The problem is that adversaries can react several rounds faster than honest participants.[1] We first present it in the synchronous model and then discuss why it is not only a technical difficulty of this model.

If an honest user sends a message using the real encryption system, the message goes via the sender's machine and the recipient's machine. The corresponding timing must be modeled in the trusted host, i.e., the trusted host delivers messages after two switching steps. Thus, with the trusted host, an honest user cannot get an answer referring to his message earlier than four rounds after his message. However, if the recipient in the real system is corrupted, he can save the two rounds where his own machine would handle the message by taking it immediately from the line, decrypting it (he knows his own decryption key), and composing and encrypting the answer, all in one round. Thus an answer from a dishonest recipient can arrive two rounds faster. We see no realistic way to improve the real system so that it corresponds to the same trusted host for honest and dishonest recipients in this respect. We therefore model this tolerable imperfection in the trusted host by offering a faster service at the ports for the adversary, so that anyone basing an application on the trusted host must be aware of it.

It is reasonable to ask whether this problem would disappear in asynchronous models and whether those aren't more realistic. (Or one can relax the timing requirements in the comparison, i.e., not require that events in the ideal and the real system must happen in

[1]This is different from the problem of rushing adversaries within a round, which is already taken care of in the synchrony model.

the same rounds, as in [30].) First, however, many cryptographic protocols are designed for synchronous systems and it should be possible to define their security.[2] Secondly, in real life the problem occurs whenever the users (not the machines of the system!) have access to real time. This seems unavoidable because we cannot define that cryptographic systems must never be used in real-time applications or that human users must not look at their watches.[3] Thus, in a real system an adversary may indeed be faster (e.g., by bridging network delays), and honest users can notice this. This will usually not have a very bad effect, but contradicts real-life indistinguishability of the real and ideal system. Hence we believe that the timing differences must be modeled in the trusted host. One can even construct artificial examples where users tell secrets to someone who can react very fast, e.g., because they believe that the person must have known the secrets before; such a problem might occur in a badly designed synchronous protocol for mutual identification. Another problem is timing channels in the classical sense. For example, consider a user who is known to answer all good news fast and all bad news slowly. An adversary in the real system can, simply by the timing of the traffic pattern, judge whether this user gets good or bad news, but this imperfection is not visible in an asynchronous model. Concrete examples of this type might be automatic message-processing systems that answer certain types of requests faster than others.

Even more classical timing channels are those where the users are considered dishonest, e.g., a Trojan horse that tries to send secret information out from an infected program to its creator, but without having subverted the operating system. Our model, with a universal quantifier over honest users H, also includes those that are actually two machines H_1 and H_2 without connection, and requires that they cannot communicate better using the real system than in the ideal system.

3.2 An Ideal System

We now define an ideal system that models secure message transmission with three tolerable imperfections: observable traffic patterns, suppressible messages, and faster service for the adversary. As discussed in the previous subsection, this is the optimum that can be achieved without a large increase in the network traffic.

The ideal system is of the standard cryptographic type described at the end of Section 2.5.

Scheme 3.1 (Ideal System for Secure Message Transmission) *Let a polynomial-time decidable set $Msg \subseteq \{0,1\}^{\leq len}$ of messages of length at most len be given with $\epsilon \in Msg$, where ϵ is the empty word and stands for "no message". Let a number $n \in \mathbb{N}$ of intended*

[2]Of course, the synchrony is typically virtual, i.e., derived from loosely synchronized clocks and bounds on message delays after which a message will be considered lost.

[3]The rounds in a cryptographic protocol used over the Internet may have to be quite long to tolerate temporary congestion or message retransmission. Hence they may indeed be observable with watches.

participants be given and $\mathcal{M} := \{1, \ldots, n\}$, and let $R \in \mathbb{N}$ be the intended number of rounds. An ideal system for secure message transmission is then defined as

$$Sys^{\mathsf{secmsg,id}}_{Msg,n,R} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) | \mathcal{H} \subseteq \mathcal{M}\},$$

(i.e., the access structure is the powerset of $\mathcal{M}$), where $\mathsf{TH}_{\mathcal{H}}$ and $S_{\mathcal{H}}$ are defined as follows. Let $\mathcal{A} = \mathcal{M} \setminus \mathcal{H}$ be the set of corrupted participant indices. We often write "user u" or "participant u" although u is only the index.

Ports: *The specified ports of $\mathsf{TH}_{\mathcal{H}}$ are $S_{\mathcal{H}} := \{\mathsf{in}_u?, \mathsf{out}_u! | u \in \mathcal{H}\}$ and the unspecified ports $\bar{S}_{2,\mathcal{H}} = \{\mathsf{adv\_out!}, \mathsf{busy!}, \mathsf{adv\_in?}, \mathsf{suppress?}\}$; thus $Ports_{\mathsf{TH}_{\mathcal{H}}} = S_{\mathcal{H}} \cup \bar{S}_{2,\mathcal{H}}$ (see Figure 3.1).*

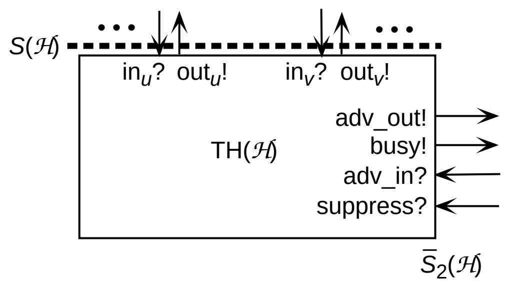

Figure 3.1: A structure of the ideal system for secure message transmission

The internal state of $\mathsf{TH}_{\mathcal{H}}$ consists of the current round number i, which it updates with every transition, and a matrix $in_{i-1} \in Msg^{\mathcal{M}^2}$. In Round 0, $\mathsf{TH}_{\mathcal{H}}$ does nothing. (This is time reserved for initialization in the real system.) Now we consider the state transition for any Round $i > 0$.

Inputs: *At each port $\mathsf{in}_s?$, $\mathsf{TH}_{\mathcal{H}}$ expects an input vector $(in_{i,s,r})_{r \in \mathcal{M}} \in Msg^{\mathcal{M}}$. By "expects" we mean that it replaces any other input by a vector of all ϵ; similarly for other structures in the following. (Hence every user s may send one message to every other user r in each round.) At the ports $\mathsf{adv\_in}?$ and $\mathsf{suppress}?$, it expects matrices $adv\_in_i \in Msg^{\mathcal{A}\times\mathcal{H}}$ and $suppress_i \in \{0,1\}^{\mathcal{H}^2}$.*

Computation: *For all $s, r \in \mathcal{H}$, the trusted host sets*

$$out_{i,s,r} := \begin{cases} \epsilon & \text{if } suppress_{i,s,r} = 1 \\ in_{i-1,s,r} & \text{else;} \end{cases} \tag{3.1}$$

$$busy_{i,s,r} := \begin{cases} 1 & \text{if } in_{i,s,r} \neq \epsilon \\ 0 & \text{else.} \end{cases} \tag{3.2}$$

Hence it delays messages between honest users for one round, but immediately tells the adversary where such messages are in transit. For $a \in \mathcal{A}$, $u \in \mathcal{H}$, it sets

$$out_{i,a,u} := adv\_in_{i,a,u}; \tag{3.3}$$

$$adv\_out_{i,u,a} := in_{i,u,a}. \tag{3.4}$$

Hence messages to and from the adversary are delivered immediately.

Outputs: *The matrices $adv\_out_i \in Msg^{\mathcal{H} \times \mathcal{A}}$ and $busy_i \in \{0,1\}^{\mathcal{H}^2}$ are output at the ports* $\mathsf{adv\_out!}$ *and* busy*, and each tuple $(out_{i,s,r})_{s \in \mathcal{M}}$ at the port* $\mathsf{out}_r!$.

◇

Fixing a finite message space is needed because otherwise the length of a message cannot be secret. The bound on the number of rounds is not essential, but simplifies the representation and is no serious restriction. The input and output vectors at the specified ports, i.e., for honest users, could be implemented with a (fixed) compressed representation because they will usually be sparse. The representation of the adversary in- and outputs is irrelevant because they only occur in the ideal system.

3.3 A Real System

We now define a real system for secure message transmission. The main part, composition and decomposition of network messages, is Equations 3.5 and 3.6.

We use asymmetric encryption and digital signatures. Let Msg be the message space, R the number of rounds, and $\mathcal{M} = \{1, \ldots, n\}$ as in Scheme 3.1.

The algorithms $(\mathsf{gen_S}, \mathsf{sign}, \mathsf{test})$ denote a secure digital signature scheme [18, 27] whose message space includes $Msg_S := Msg \times \mathcal{M} \times \{1, \ldots, R\}$ (for all security parameters).[4] We use slightly abbreviated notation: We write $(\mathsf{sign}_u, \mathsf{test}_u) \leftarrow \mathsf{gen_S}(1^k)$ for the generation of a signing key and a test key based on a security parameter k. By $sig \leftarrow \mathsf{sign}_u(m)$, we denote a signature on the message $m \in Msg_S$, including m itself.[5]We denote the resulting signature space by $Sig_S(k)$. The length of the signatures is polynomially bounded in k. The verification $\mathsf{test}_u(sig)$ returns m if the signature is valid with respect to the included message, else false.

By $(\mathsf{gen_E}, \mathsf{E}, \mathsf{D})$, we denote an encryption scheme secure against adaptive chosen-ciphertext attacks, e.g., [16]. We write $(\mathsf{E}_u, \mathsf{D}_u) \leftarrow \mathsf{gen_E}(1^{k^*(k)})$ for the generation of an encryption key and a decryption key. Here, $k^*(k)$ denotes a security parameter that allows us to securely encrypt the message space $\mathcal{M} \times Sig_S(k)$ for the given k.[6] We denote the (probabilistic) encryption of a message m by $c \leftarrow \mathsf{E}_u(m)$, and decryption by $m \leftarrow \mathsf{D}_u(c)$.

The comma denotes tuple composition, not concatenation, and its implementation must guarantee unambiguous decomposition.

[4]We repeat the security definitions in more detail in Section 3.5. If the original scheme has a too small message space, a combination with a collision-free hash function retains security [17].

[5]We even assume that it is a pair of the message and the actual signature which can be uniquely decomposed. Then sig uniquely determines m independently of the key. Otherwise more complicated message formats would be needed below, which is a waste of bandwidth in the standard case.

[6]We do not require all input messages to be of the same length. However, security for this message space implies that also the message length is hidden by the encryption.

Scheme 3.2 (Real System for Secure Message Transmission) *Let a message set Msg, a set $\mathcal{M} = \{1, \ldots, n\}$ of participant indices, and a round number R be given as in Scheme 3.1. The system is a standard cryptographic system according to Definitions 2.12 and 2.13. Hence we only have to define the intended structure $(\hat{M}^*, S^*)$ and the trust model, but for readability of the proof, we also describe the resulting actual structures. $\hat{M}^*$ is a set $\{\mathsf{M}_u | u \in \mathcal{M}\}$ and the access structure is the powerset of $\mathcal{M}$. Hence the system is of the form*

$$Sys^{\mathsf{secmsg,real}}_{Msg,n,R} = \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) | \mathcal{H} \subseteq \mathcal{M}\}$$

with $\hat{M}_{\mathcal{H}} = \{\mathsf{M}_{u,\mathcal{H}} | u \in \mathcal{H}\}$.

Ports: *The ports of machine M_u are $\{\mathsf{in}_u?, \mathsf{out}_u!\} \cup \{\mathsf{net}_{u,v}!, \mathsf{net}_{v,u}? | v \in \mathcal{M}\} \cup \{\mathsf{aut}_{u,v}!, \mathsf{aut}_{v,u}? | v \in \mathcal{M}\}$. The first ones are for the user, the second ones for normal messages to and from each M_v, and the last ones for key exchange, only used in Round 0.*

The specified ports are $S^* := \{\mathsf{in}_u?, \mathsf{out}_u! | u \in \mathcal{M}\}$.

Channel model χ: *Connections $\{\mathsf{net}_{u,v}!, \mathsf{net}_{u,v}?\}$ are insecure and connections $\{\mathsf{aut}_{u,v}!, \mathsf{aut}_{u,v}?\}$ authentic. The resulting ports in a real structure are illustrated in Figure 3.2.*

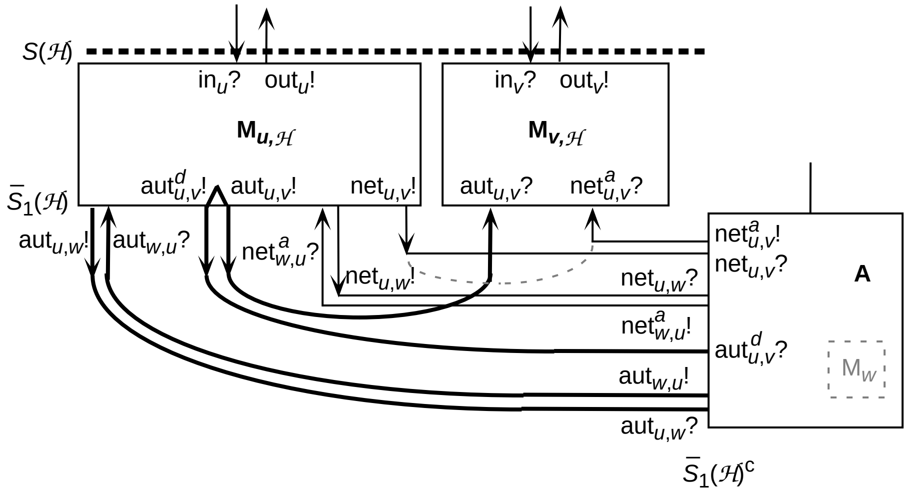

Figure 3.2: Classes of ports in the real system for secure message transmission. Two correct machines are shown and what became of the intended connections from $\mathsf{M}_{u,\mathcal{H}}$ to $\mathsf{M}_{v,\mathcal{H}}$ (port names in the machines) and between $\mathsf{M}_{u,\mathcal{H}}$ and a machine M_w with $w \notin \mathcal{H}$ (port names outside). Authentic connections are bold, others normal, and the original of a modifiable connection is dashed.

The state-transition function of a machine $\mathsf{M}_{u,\mathcal{H}}$ is defined as follows:

Round 0 (Initialization) *It generates two key pairs, $(\mathsf{sign}_u, \mathsf{test}_u) \leftarrow \mathsf{gen_S}(1^k)$ and $(\mathsf{E}_u, \mathsf{D}_u) \leftarrow \mathsf{gen_E}(1^{k^*(k)})$, and outputs $(\mathsf{test}_u, \mathsf{E}_u)$ at $\mathsf{aut}_{u,v}!$ and $\mathsf{aut}^d_{u,v}!$ for all $v \in \mathcal{M}$.*

Now we consider $\mathsf{M}_{u,\mathcal{H}}$ in an arbitrary round $i > 0$.

Inputs: *It expects an input vector $(in_{i,u,r})_{r\in\mathcal{M}} \in Msg^{\mathcal{M}}$ at $\mathsf{in}_u?$. If $i = 1$, it also expects a pair $(\mathsf{test}_v, \mathsf{E}_v)$ (of public keys) at each port $\mathsf{aut}_{v,u}?$. If $i > 1$, it reads a value $net^a_{i-1,s,u}$ from each port $\mathsf{net}^a_{s,u}?$.*

Computation: *It composes a* network message *for each recipient $r \in \mathcal{M}$ as*

$$net_{i,u,r} \leftarrow \begin{cases} \mathsf{E}_r(u, \mathsf{sign}_u(in_{i,u,r}, i, r)) & \textit{if } in_{i,u,r} \neq \epsilon, \\ \epsilon & \textit{else.} \end{cases} \tag{3.5}$$

We abbreviate this function by $net_{i,u,r} \leftarrow \mathsf{comp}(i, u, r, in_{i,u,r})$.

If $i > 1$, it decomposes each received message $net^a_{i-1,s,u}$ as

$$(s_{i-1,s,u}, sig_{i-1,s,u}) \leftarrow \mathsf{D}_u(net^a_{i-1,s,u}).$$

If the decryption or decomposition fails, let both components be ϵ. Then

$$out_{i,s,u} := \begin{cases} m & \textit{if } s_{i-1,s,u} = s \wedge \mathsf{test}_s(sig_{i-1,s,u}) = (m, i-1, u) \\ \epsilon & \textit{else.} \end{cases} \tag{3.6}$$

We abbreviate this function by $out_{i,s,u} \leftarrow \mathsf{decomp}(i, s, u, net^a_{i-1,s,u})$.

Outputs: *It outputs the vector $(out_{i,s,u})_{s\in\mathcal{M}}$ at $\mathsf{out}_u!$, and each value $net_{i,u,r}$ at $\mathsf{net}_{u,r}!$.*

◇

3.4 Remarks on the Message Format

The format of the network message may look complicated. However, simpler formats are not always secure. In particular, omitting the identity u in the encryption, i.e.,

$$net_{i,u,r} \leftarrow \mathsf{E}_r(\mathsf{sign}_u(in_{i,u,r}, i, r))$$

is insecure although one may feel that the signature inside determines the identity: An adversary can choose one of his own keys test_a equal to one of a correct machine, say test_s, because M_s switches in Round [0.1] and A in Round [0.2]. Later it can take a network message $net_{i,s,r}$ sent between two honest participants and use it as its own network message $net^a_{i,a,r}$ (again because A switches after M_s in Round i). This message passes the test, and thus $in_{i,s,r}$ is output to H as $out_{i+1,a,r}$. An adversary in the ideal system cannot achieve this effect. It is even dangerous in practice, because H, believing that it obtained this message from M_a, might freely send parts of it back to M_a in a reply.

This attack on the simplified network messages could be avoided by verifying that all public keys are different. However, this would not imply provable security given the normal definition of a signature system (see Definition 3.1): It is not excluded that an adversary can choose a key related to the key of a correct machine such that signatures made with sign_s are also acceptable with respect to test_a.

First encrypting and then signing does not automatically work either, e.g.,

$$net_{i,u,r} \leftarrow \mathsf{sign}_u(i, r, \mathsf{E}_r(in_{i,u,r}))$$

has the same problem even more obviously: The adversary can take the ciphertext $c = \mathsf{E}_r(in_{i,u,r})$ from such a message and also send it in a message of his own as $net^a_{j,a,r} \leftarrow \mathsf{sign}_a(j, r, c)$.

The inclusion of the round number i is necessary for freshness. An alternative would be nonces, but this requires a multi-round protocol for each message.

3.5 Security Proof

We now show that the real system is as secure as the ideal system. Our simulation is blackbox, see Definition 2.10. We use Corollary 2.3, i.e., we only consider users that use precisely the specified ports. Hence the set P of ports of the real adversary always comprises $\bar{S}^c_{1,\mathcal{H}}$.

Scheme 3.3 (Simulator) *Let a set $\mathcal{H} \subseteq \mathcal{M}$ of (indices of) correct participants be given and $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$. Let $P = \bar{S}^c_{1,\mathcal{H}} \cup P'$ be a set of adversary ports.*

The definition of the simulator $\mathsf{Sim}_{\mathcal{H},P}(\mathsf{A})$ is illustrated in Figure 3.3. If $\mathcal{H}$ and P are clear from the context, we write $\mathsf{Sim}(\mathsf{A})$. Its ports must be as in Figure 3.1, here $Ports_{\mathsf{Sim(A)}} = \bar{S}^c_{2,\mathcal{H}} \cup P'$. It leaves the communication between A and H unchanged. In slight abuse of notation we therefore say Sim for the part of $\mathsf{Sim(A)}$ without A; then $Ports_{\mathsf{Sim}} = \bar{S}_{1,\mathcal{H}} \cup \bar{S}^c_{2,\mathcal{H}}$ corresponding to Figure 3.2.

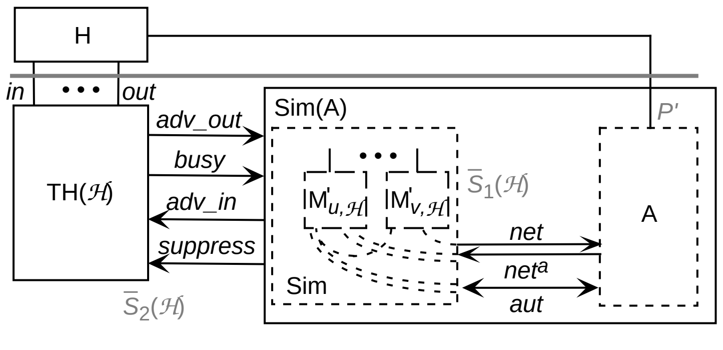

Figure 3.3: Set-up of the simulation. The names on the arrows are names of message classes (slightly summarizing the port names).

Internally, Sim more or less simulates the real machines; we illustrate this by $\mathsf{M}'_{u,\mathcal{H}}$ for $u \in \mathcal{H}$. The timing of the main message sequence and its simulation is illustrated in

Figures 3.4 and 3.5. We define the transitions of Sim(A) *as the compressed version of a 6-subround clocking scheme* $\kappa_6 = (\{\mathsf{TH}_\mathcal{H}, \mathsf{H}\}, \mathsf{Sim}, \mathsf{A}, \mathsf{H}, \mathsf{A}, \mathsf{Sim})$ *where Subrounds 2 with 3 and 5 with 6 are joined to get the correct clocking for the ideal system (see Lemma 2.2d). We call the joined subrounds 2a and 2b, and 4a and 4b.*

Essentially, in Subround $[i.2a]$, Sim *transforms the abstract messages it got from* $\mathsf{TH}_\mathcal{H}$ *into suitable network messages for* A, *and in Subround* $[i.4b]$ *it transforms the messages from* A *into corresponding signals to* $\mathsf{TH}_\mathcal{H}$.

Round $[0.2a]$ Sim *simulates the key generation of the machines* $\mathsf{M}_{u,\mathcal{H}}$ *for* $u \in \mathcal{H}$ *without changes.*

Round $[0.4b]$ Sim *stores received keys as in Round* $[1.1]$ *of the real system.*

Round $[i.2a]$ **for** $i > 0$**:** Sim *obtains matrices* adv_out_i *and* $busy_i$ *from* $\mathsf{TH}_\mathcal{H}$.

Each element $adv\_out_{i,s,r}$ *represents a message from a correct sender* $s \in \mathcal{H}$ *for a corrupted recipient* $r \in \mathcal{A}$. *Here* Sim *computes the corresponding network message as* $net_{i,s,r} \leftarrow \mathsf{comp}(i, s, r, adv\_out_{i,s,r})$.

Each element $busy_{i,s,r} = 1$ *signals a message between correct machines* $s, r \in \mathcal{H}$. *Here* Sim *computes* $net_{i,s,r} \leftarrow \mathsf{comp}(i, s, r, m_{sim})$ *with a fixed message* $m_{sim} \in Msg \setminus \{\epsilon\}$. *(It cannot obtain the actual input made at the port* $\mathsf{in}_s?$ *because* $\mathsf{TH}_\mathcal{H}$ *keeps it secret.)*

Round $[i.4b]$ **for** $i > 0$**:** *Now* Sim *converts the messages* $net^a_{i,s,r}$ *received from* A *(in Rounds* $[i.2b]$ *and* $[i.4a]$*) into inputs to* $\mathsf{TH}_\mathcal{H}$:

For $s \in \mathcal{A}$ *and* $r \in \mathcal{H}$, *it sets* $adv\_in_{i+1,s,r} := \mathsf{decomp}(i+1, s, r, net^a_{i,s,r})$.

For $s, r \in \mathcal{H}$ *and if* $busy_{i,s,r} = 1$, *the trusted host expects to receive* $suppress_{i+1,s,r}$ *indicating whether the message is destroyed in transit. Therefore* Sim *sets* $m := \mathsf{decomp}(i+1, s, r, net^a_{i,s,r})$. *If* $m = \epsilon$, *it sets* $suppress_{i+1,s,r} := 1$, *otherwise* $suppress_{i+1,s,r} := 0$. *If* $m \neq m_{sim} \wedge m \neq \epsilon$, *or* $m \neq \epsilon \wedge busy_{i,s,r} = 0$, *the simulator stops.*[7]

◇

For the following security proof, we need the underlying definitions of secure encryption and signature schemes. Security of a signature scheme means that existential forgery is infeasible even in adaptive chosen-message attacks [27].

Definition 3.1 (Security of Signature Schemes) *An arbitrary (probabilistic) polynomial-time machine* $\mathsf{A}_{\mathsf{sig}}$ *interacts with a signer machine* Sig *(also called signing oracle) defined as follows:*

[7] In the real system, m would be output at $\mathsf{out}_r!$, but the simulator has no way of making $\mathsf{TH}_\mathcal{H}$ do this; this is where message integrity is expressed in $\mathsf{TH}_\mathcal{H}$. Hence the simulation would become distinguishable and can as well be stopped. The proof will show that this case only occurs if A has forged a signature, and thus it is negligible.

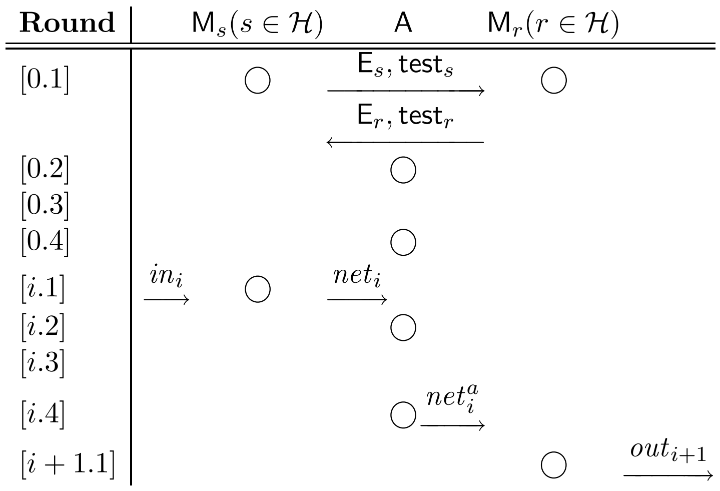
Round
$\mathsf{M}_s(s \in \mathcal{H})$
A
$\mathsf{M}_r(r \in \mathcal{H})$
[0.1]
$\mathsf{E}_s, \mathsf{test}_s$
$\mathsf{E}_r, \mathsf{test}_r$
[0.2]
[0.3]
[0.4]
[i.1]
in_i
net_i
[i.2]
[i.3]
[i.4]
net_i^a
[$i+1$.1]
out_{i+1}

Figure 3.4: Timing of Scheme 3.2 for key exchange and a message between two correct machines. ◯ denotes relevant switching of the machine in this column; H switches in [i.1] and [i.3].

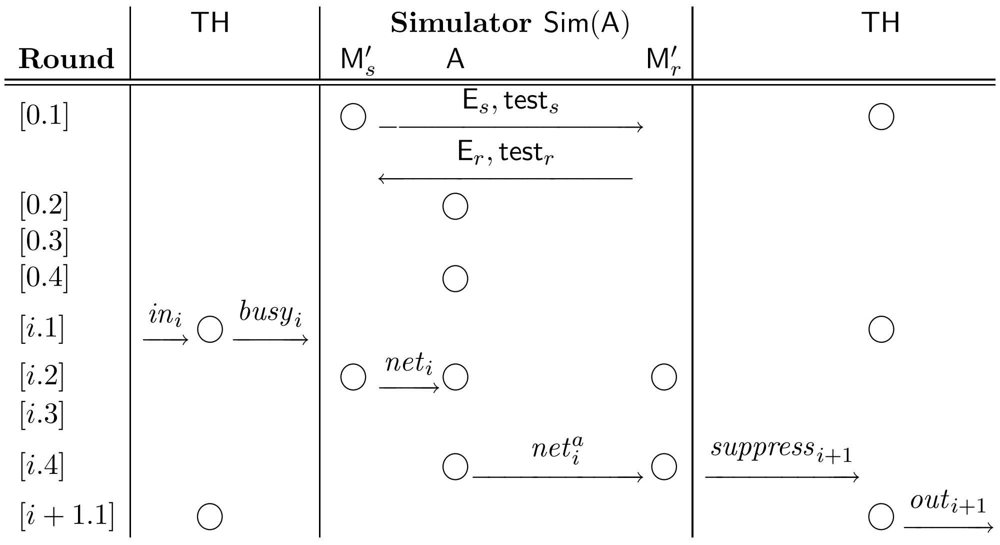

Figure 3.5: Timing of Scheme 3.3 for the same situation as in Figure 3.4

1. Sig *generates a key pair,* $(\mathsf{sign}, \mathsf{test}) \leftarrow \mathsf{gen}_\mathsf{S}(1^k)$*, and sends* test *to* A_sig.

2. *In each round,* Sig *signs an arbitrary message* m_j *it receives from* A_sig *(automatically only a polynomial number if* A_sig *is polynomial).*

3. *Finally,* A_sig *should output a value sig.*

A_sig *has won if* $\mathsf{test}(sig)$ *gives a message* m *with* $m \neq m_j$ *for all* j*, i.e., sig is a valid signature on a message that* Sig *did not sign. The probability of this event must be negligible in* k*. (In our terminology, we have a closed collection of 2 machines clocked alternately, and the event is a predicate on the runs; hence the probability is well-defined.)* $\diamond$

Security of an encryption scheme means that any two messages must be indistinguishable even in adaptive chosen-ciphertext attacks. This was introduced in [51], used for an efficient construction in [16] and formalized as "IND-CCA2" in [9].

Definition 3.2 (Security of Encryption Schemes) *An arbitrary (probabilistic) polynomial-time machine* A_enc *interacts with a decryptor machine* Dec *(also called decryption oracle) defined as follows:*

1. Dec *generates a key pair,* $(\mathsf{E}, \mathsf{D}) \leftarrow \mathsf{gen}_\mathsf{E}(1^k)$*, and sends* E *to* A_enc.

2. *In each round,* Dec *decrypts an arbitrary ciphertext* c_j *received from* A_enc.

3. *At some point,* A_enc *sends a pair* (m_0, m_1) *of messages to* Dec*. Then* Dec *randomly chooses a bit* b *and returns an encryption* $c \leftarrow \mathsf{E}(m_b)$.

4. A_enc *may again ask* Dec *to decrypt ciphertexts* c_j*, but now* Dec *refuses to decrypt if* c_j *equals its own ciphertext* c.

5. *Finally,* A_enc *should output a bit* b^*.

This bit is meant as a guess at b*, i.e., which of the two messages is contained in* c*. The probability of the event* $b^* = b$ *must be bounded by* $1/2 + 1/\mathsf{poly}(k)$*. (Again this is a 2-machine collection and thus the probability is well-defined.)* $\diamond$

Theorem 3.1 (Security of the Message Transmission System) *For any message set Msg, index set* $\mathcal{M} = \{1, \ldots, n\}$ *and round number* R *as in Scheme 3.1,*

$$Sys^{\mathsf{secmsg,real}}_{Msg,n,R} \geq^{f,\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{secmsg,id}}_{Msg,n,R}$$

for the canonical mapping f *from Section 2.5 if the signature and encryption schemes used in Scheme 3.2 are secure.* $\square$

Recall that the canonical mapping f maps each real structure $(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})$ to the ideal structure $(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}})$ for the same set $\mathcal{H}$.

Proof. Let a set $\mathcal{H}$ and thus a structure $(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})$ and a port set $P = \bar{S}^c_{1,\mathcal{H}} \cup P'$ be given. In the following, we omit the parameter $\mathcal{H}$ of all machines and write $\mathsf{Sim}(\mathsf{A})$ for $\mathsf{Sim}_P(\mathsf{A})$. Now let a configuration $conf_1 = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}^f(Sys^{\mathsf{secmsg,real}}_{Msg,n,R})$ with $Ports_{\mathsf{A}} = P$ be given. By Remark 2.7, we can assume that H is not clocked in Subrounds $[i.1]$. We claim that $conf_2 = (\{\mathsf{TH}\}, S, \mathsf{H}, \mathsf{Sim}(\mathsf{A})) \in Indist^f(conf_1)$, i.e., $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$.

We show the stronger statement $view_{conf_1}(\mathsf{H}, \mathsf{A}) \approx view_{conf_2}(\mathsf{H}, \mathsf{A})$.Intuitively this means that we compare $\hat{M}$ and the combination of TH and Sim. For this, we first have to show that the clocking makes no difference. Then, there are intuitively two aspects in the simulation that make it only computationally indistinguishable: a message m_{sim} is often encrypted instead of a real message, and a simulation may stop prematurely due to what should be a successful signature forgery by A. We therefore show by two reduction proofs that we could break one of the underlying schemes if the views were distinguishable.

Clocking: We defined $\mathsf{Sim}(\mathsf{A})$ in $conf_2$ as a combination based on a collection where Sim and A are separate (we now call it $conf_2^6$) and a 6-subround clocking scheme κ_6. Hence $view_{conf_2}(\mathsf{H}, \mathsf{A})$ equals $view_{conf_2^6}(\mathsf{H}, \mathsf{A})$ except for subround renaming. In $conf_2^6$, only Sim and TH are clocked in Subrounds 4b, 1, and 2a. Let $\mathsf{TH} + \mathsf{Sim}$ denote the hiding combination of TH and Sim according to Lemma 2.2 and where these subrounds are joined. Then $conf_2^* = (\mathsf{TH}{+}\mathsf{Sim}, S, \mathsf{H}, \mathsf{A})$ is a configuration with the standard clocking scheme, and $view_{conf_2^*}(\mathsf{H}, \mathsf{A})$ equals $view_{conf_2^6}(\mathsf{H}, \mathsf{A})$ except for the subround renaming.In both renamings of $conf_2^6$, H ends up in Subround 3 and A in 2 and 4. Hence

$$view_{conf_2^*}(\mathsf{H}, \mathsf{A}) = view_{conf_2}(\mathsf{H}, \mathsf{A}).$$

Signatures: We first show that the probability that the simulation stops prematurely is negligible. Assume the contrary. We then construct an adversary $\mathsf{A}_{\mathsf{sig}}$ against the signature scheme using H and A as blackboxes (recall Definition 3.1) as shown in Figure 3.6. $\mathsf{A}_{\mathsf{sig}}$ randomly chooses $s^* \in \mathcal{H}$ and starts simulating $\mathsf{TH}{+}\mathsf{Sim}$ interacting with H and A. It does everything as $\mathsf{TH} + \mathsf{Sim}$ would except that it uses the public key test from Sig as test_{s^*}. Thus whenever it has to execute $sig_j \leftarrow \mathsf{sign}_{s^*}(m_j)$ for some message m_j, it sends m_j to Sig instead and uses the answer as sig_j.[8] If the simulation stops prematurely in a subround $[i+1.1]$, this corresponds to Subround $[i.4b]$ in the original clocking of Sim, and A has sent a network message $net^a_{i,s,r}$ for some $s, r \in \mathcal{H}$ that is correct according to Equation (3.6) but contains a message $m' \neq m_{sim}$, or any $m' \neq \epsilon$ although $busy_{i,s,r} = 0$. Then if $s = s^*$, the machine $\mathsf{A}_{\mathsf{sig}}$ outputs the second component sig of $\mathsf{D}_r(net^a_{i,s,r})$.

We first show that if $\mathsf{A}_{\mathsf{sig}}$ makes any output, it is a successful forgery. By Equation (3.6), $\mathsf{test}_{s^*}(sig) = (m', i, r) =: m^*$. We have to show that $m^* \neq m_j$ for all messages m_j that $\mathsf{A}_{\mathsf{sig}}$

[8]There is no clocking problem although we defined that Sig signs only one message per round because $\mathsf{A}_{\mathsf{sig}}$ can clock its submachines itself, i.e., the rounds indexed j and those indexed i are different.

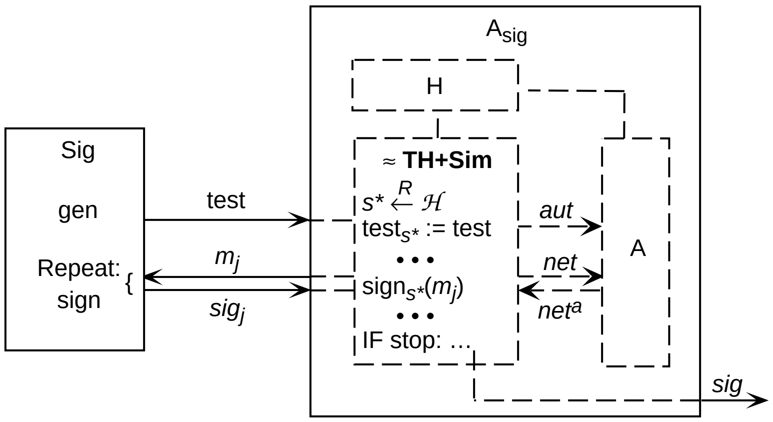

Figure 3.6: Reduction to an adversary on the signature scheme

asked Sig to sign. Assume the contrary. By construction, messages m_j only arise when Sim applies Equation (5) to construct a message $net_{i',s^*,r'}$ in an original subround $[i'.2a]$, now $[i'.1]$, and then $m_j = (in', i', r')$ for some in'. By the required unambiguous decomposition of the representation of message triples, $m^* = m_j$ implies $in' = m' \wedge i' = i \wedge r' = r$. As $r \in \mathcal{H}$, the construction of Sim implies $m' = m_{sim}$, and actually m_j is only signed if $busy_{i,s^*,r} = 1$. This contradicts the preconditions on m' (as a message leading to premature stopping).

Secondly, we show that the probability that $\mathsf{A_{sig}}$ makes an output is not negligible. $\mathsf{A_{sig}}$ combined with Sig behaves exactly like $\mathsf{TH} + \mathsf{Sim}$ until the stopping time. Hence the probability of stopping in the simulation equals that in $conf_2^*$. If stopping occurs, the stopping conditions are fulfilled for one $s \in \mathcal{H}$. As $\mathsf{A_{sig}}$ chose s^* randomly and no information about s^* is visible to H and A, the probability that $s^* = s$ is $|\mathcal{H}|^{-1}$.[9] Hence the success probability of $\mathsf{A_{sig}}$ is a fixed fraction $|\mathcal{H}|^{-1}$ of the probability that Sim stops prematurely, and therefore also not negligible. This is the desired contradiction with the security of the signature scheme.

As a consequence, we can modify the behaviour of $\mathsf{TH} + \mathsf{Sim}$ in the case that Sim stops prematurely: We define a machine $\mathsf{TH}' + \mathsf{Sim}'$ that, instead of stopping under the given conditions, outputs the forged message m' as $out_{i+1,s,r}$ (as the real machine M_r would do in $conf_1$). Let $conf_2' = (\mathsf{TH}' + \mathsf{Sim}', S, \mathsf{H}, \mathsf{A})$. As a difference only occur with negligible probability, we have shown that[10]

$$view_{conf_2'}(\mathsf{A}, \mathsf{H}) \approx view_{conf_2^*}(\mathsf{A}, \mathsf{H}).$$

[9]More formally, one transforms the given probability space, where s^* is chosen at the start, such that s^* is chosen independently at the end, exploiting that the rest of the runs is compatible with any s^*.

[10]In more detail, we have shown that the change only concerns a negligible fraction of the runs. Hence the runs of the two configurations are statistically indistinguishable for the class *NEGL*. (Note that they are only of polynomial length.) This implies statistical and thus also computational indistinguishability of all functions of the runs, in particular the views, by Lemma 2.3.

As a technicality for the following proof, we need a similar statement for the correct machines. Let M' be a machine that acts like the hiding combination of $\hat{M}$ with one exception: If $\mathsf{decomp}(i, s, r, net^a_{i-1,s,r})$ yields $m = m_{sim} \neq in_{i-1,s,r}$, then M' sets $out_{i,s,r} = in_{i-1,s,r}$ instead of m. The proof is a reduction just as above: We have a forged signature on $(m_{sim}, i-1, r)$ because only $(in_{i-1,s,r}, i-1, r)$ was signed with the components $i-1$ and r. Let $conf'_1 = (\mathsf{M}', S, \mathsf{H}, \mathsf{A})$. Hence we have

$$view_{conf'_1}(\mathsf{A}, \mathsf{H}) \approx view_{conf_1}(\mathsf{A}, \mathsf{H}).$$

Encryption, hybrid argument: The encryption part is proven with a reactive version of the typical "hybrid arguments", e.g., known from [26]. Intuitively, we show that to distinguish the overall views, one has to distinguish at least one particular encryption. (Hence this part of the proof also shows that all other aspects of the simulation are correct.) For this, we define a hybrid machine Hyb_t for each triple $t = (i, s, r) \in \{1, \ldots, R\} \times \mathcal{H}^2$. Roughly, it treats the inputs $in_{i',s',r'}$ for $s', r' \in \mathcal{H}$ as in the real system up to the triple (i, s, r) and afterwards as in the simulation.[11] We write "$\leq$" for the lexicographic order on these triples, t_{max} for their maximum, and $\mathsf{pred}(t)$ for the predecessor of t in this order. Furthermore, let $0 < t$ for all these triples.

We define $\mathsf{Hyb}_0 := \mathsf{TH}' + \mathsf{Sim}'$. For $t = (i, s, r)$, the key exchange of Hyb_t is as in both M' and $\mathsf{TH}' + \mathsf{Sim}'$. We now have to show how Hyb_t computes all values $net_{i',s',r'}$ and $out_{i',s',r'}$.

1. For $(i', s', r') \leq t$, it computes $net_{i',s',r'} \leftarrow \mathsf{comp}(i', s', r', in_{i',s',r'})$ like $\hat{M}$ and thus M'.

2. For $(i', s', r') > t$, it computes $net_{i',s',r'}$ like $\mathsf{TH}' + \mathsf{Sim}'$:

    a) For $r' \in \mathcal{H}$, this means $net_{i',s',r'} = \epsilon$ if $in_{i',s',r'} = \epsilon$ and thus $busy_{i',s',r'} = \epsilon$; otherwise $net_{i',s',r'} \leftarrow \mathsf{comp}(i', s', r', m_{sim})$.

    b) For $r' \in \mathcal{A}$, it means that $adv\_out_{i',s',r'} = in_{i',s',r'}$ and thus $net_{i',s',r'} \leftarrow \mathsf{comp}(i', s', r', in_{i',s',r'})$ (as in M').

3. For $(i'-1, s', r') \leq t$, it computes $out_{i',s',r'}$ like M', i.e., as $\mathsf{decomp}(i', s', r', net^a_{i'-1,s',r'})$ except that $in_{i'-1,s',r'}$ is output instead if the result is m_{sim}.

4. For $(i'-1, s', r') > t$, it computes $out_{i',s',r'}$ like $\mathsf{TH}' + \mathsf{Sim}'$: It first sets $m = \mathsf{decomp}(i', s', r', net^a_{i'-1,s',r'})$.

    a) If $s' \in \mathcal{H}$:

        – If $m \neq \epsilon$ and $m \neq m_{sim} \vee busy_{i'-1,s',r'} = 0$, then $out_{i',s',r'} = m$ (here $\mathsf{TH}' + \mathsf{Sim}'$ differs from $\mathsf{TH} + \mathsf{Sim}$).

[11] It is not trivial to define hybrid reactive systems generally: If the configurations are probabilistic and have memory, it may not be clear how to initialize the memory of configuration 2 such that it is consistent with the execution of configuration 1 so far. Hence we make the particular construction explicit.

– If $m = m_{sim} \wedge busy_{i'-1,s',r'} = 1$, then $suppress_{i',s',r'} = 0$ and thus $out_{i',s',r'} = in_{i'-1,s',r'}$.

– If $m = \epsilon$, then $suppress_{i',s',r'} = 1$ and therefore $out_{i',s',r'} = \epsilon$.

b) If $s' \in \mathcal{A}$, then m becomes $adv\_in_{i',s',r'}$ and thus $out_{i',s',r'}$ (as in M').

Clearly, $\mathsf{Hyb}_{t_{max}} = \mathsf{M}'$. As Cases 2b and 4b are as in M', each Hyb_t only differs from $\mathsf{Hyb}_{\mathsf{pred}(t)}$ in the computation of $net_{i,s,r}$ and $out_{i+1,s,r}$. Let $conf_{\mathsf{hyb},t} = (\mathsf{Hyb}_t, S, \mathsf{H}, \mathsf{A})$ for all t.

Assume that a distinguisher Dist can distinguish $view_{conf_{\mathsf{hyb},0}}(\mathsf{A}, \mathsf{H})$ and $view_{conf_{\mathsf{hyb},t_{max}}}(\mathsf{A}, \mathsf{H})$ in contradiction to Definition 2.8, i.e., the function of absolute probability differences is not negligible; we call this function $\Delta(k)$. For all t, let $p_t(k) := P(\mathsf{Dist}(1^k, view_{conf_{\mathsf{hyb},t},k}(\mathsf{A}, \mathsf{H}) = 1)$. Then

$$\left|\sum_{t=1}^{t_{max}}(p_t(k) - p_{\mathsf{pred}(t)}(k))\right| = |p_{t_{max}}(k) - p_0(k)| = \Delta(k).$$

Hence for at least one t, the sequence $(|p_t(k) - p_{\mathsf{pred}(t)}(k)|)_{k\in\mathbb{N}}$ is not negligible.[12] Let $t = (i, s, r)$ be such a triple. We assume w.l.o.g. that

$$p_t(k) - p_{\mathsf{pred}(t)}(k) > Q(k)^{-1}$$

for a polynomial Q and infinitely many k, i.e., intuitively that Dist outputs 1 to identify the real system and 0 for the simulation.[13]

Encryption, reduction: We now construct an adversary $\mathsf{A}_{\mathsf{enc}}$ against the encryption scheme (recall Definition 3.2). It uses H, A and Dist as blackboxes. The basic idea is that it simulates either Hyb_t or $\mathsf{Hyb}_{\mathsf{pred}(t)}$ depending on the bit b of the decryption oracle, i.e., without knowing which. It then uses the distinguisher to distinguish the cases, thus obtaining a guess at b. An overview is given in Figure 3.7.

1. First $\mathsf{A}_{\mathsf{enc}}$ obtains a public key E from Dec. It uses this key in the place of E_r, and generates the other keys of correct participants itself.

2. It simulates M' in interaction with the blackboxes H and A until directly before constructing $net_{i,s,r}$. (So far, M' equals both $\mathsf{Hyb}_{\mathsf{pred}(t)}$ and Hyb_t.) Where decryptions with the unknown key D_r are needed, it asks the decryption oracle Dec.

3. Now, if $in_{i,s,r} \neq \epsilon$, it sends the two messages $m_0 := (s, \mathsf{sign}_s(m_{sim}, i, r))$ and $m_1 := (s, \mathsf{sign}_s(in_{i,s,r}, i, r))$ to Dec. (Otherwise it sends nothing and sets $net_{i,s,r} = \epsilon$.) The former is the message encrypted in $\mathsf{Hyb}_{\mathsf{pred}(t)}$, the latter in Hyb_t. Hence Dec chooses $b \overset{\mathcal{R}}{\leftarrow} \{0, 1\}$ and sends a ciphertext $c \leftarrow \mathsf{E}_r(m_b)$. Then $\mathsf{A}_{\mathsf{enc}}$ uses c as $net_{i,s,r}$.

[12]The standard argument is: There is a polynomial Q with $\Delta(k) > Q(k)^{-1}$ for infinitely many k. For each of them, an index t_k with $|p_{t_k}(k) - p_{\mathsf{pred}(t_k)}(k)| > t_{max}^{-1}Q(k)$ exists. One t must occur infinitely often as t_k.

[13]Otherwise, consider the distinguisher Dist' that outputs 1 iff Dist does not.

4. When the corresponding network message $net^a_{i,s,r}$ (this is how A modified $net_{i,s,r} = c$ in transit) is handled in Round $[i + 1.1]$, $\mathsf{Hyb}_{\mathsf{pred}(t)}$ and Hyb_t also differ.

   - If $net^a_{i,s,r} = c$ (unchanged), Dec will not decrypt it, but $\mathsf{A}_{\mathsf{enc}}$ simply sets $out_{i+1,s,r} = in_{i,s,r}$. We claim that for $b = 0$ and 1, this is what $\mathsf{Hyb}_{\mathsf{pred}(t)}$ and Hyb_t, respectively, would do.
     - $b = 0$: $\mathsf{Hyb}_{\mathsf{pred}(t)}$ acts like $\mathsf{TH}' + \mathsf{Sim}'$: Sim' decrypts c to m_0, which passes all tests. Thus it obtains $m = m_{sim}$. As $in_{i,s,r} \neq \epsilon$, we had $busy_{i,s,r} = 1$ and therefore the output is $in_{i,s,r}$.
     - $b = 1$: Hyb_t acts like M'. It decrypts c to m_1 and, as this passes all tests, output its content $in_{i,s,r}$. (Even if $in_{i,s,r} = m_{sim}$, the effect is the same.)
   - If $net^a_{i,s,r} \neq c$, $\mathsf{A}_{\mathsf{enc}}$ sets $m = \mathsf{decomp}(i, s, r, net^a_{i,s,r})$, using Dec for the decryption.
     - If $m = \epsilon$, it sets $out_{i+1,s,r} = \epsilon$. This is correct for both cases.
     - If $m = m_{sim}$, it sets $out_{i+1,s,r} = in_{i,s,r}$. This is correct for M' and thus Hyb_t by definition (here M' differs from $\hat{M}$), and for $\mathsf{Hyb}_{\mathsf{pred}(t)}$ because $busy_{i,s,r} = 0$.
     - Otherwise, it sets $out_{i+1,s,r} = m$. This is correct for M' and thus Hyb_t by definition of $\hat{M}$, and for $\mathsf{Hyb}_{\mathsf{pred}(t)}$ by construction.

5. For all other messages, $\mathsf{Hyb}_{\mathsf{pred}(t)}$ and Hyb_t are the same; for concreteness let $\mathsf{A}_{\mathsf{enc}}$ simulate Hyb_t. Again, if it needs to decrypt a ciphertext c_j with D_r, it asks Dec, except if $c_j = c$. (Intuitively, this is a replay attack.) Then $\mathsf{A}_{\mathsf{enc}}$ must act on its own. The only ciphertexts Hyb_t decrypts with D_r are network messages $net^a_{i'-1,s',r}$, and it only uses the result to compute $out_{i',s',r}$. In these cases, $\mathsf{A}_{\mathsf{enc}}$ simply sets $out_{i',s',r} := \epsilon$. (Correctness is shown below.)

6. At the end, $\mathsf{A}_{\mathsf{enc}}$ inputs the view of the simulated H and A to Dist. It uses the output bit b^* of Dist as its own output.

To prove that $\mathsf{A}_{\mathsf{enc}}$, depending on b, simulates either $\mathsf{Hyb}_{\mathsf{pred}(t)}$ or Hyb_t correctly, only the correctness of $out_{i',s',r} = \epsilon$ for $net^a_{i'-1,s',r} = c$ and $(i' - 1, s') \neq (i, s)$ remains to be shown.

- $b = 0$: Then $\mathsf{Hyb}_{\mathsf{pred}(t)}$ would decrypt c to $m_0 = (s, sig)$ with $sig = \mathsf{sign}_s(m_{sim}, i, r)$. It first tests that $s = s'$. If yes, it verifies that $\mathsf{test}_s(sig) = (m', i' - 1, r)$ for some $m' \in Msg$ and the given (i', r). As we assumed that the message in clear is part of sig, this implies $i' - 1 = i$. As $(i' - 1, s') \neq (i, s)$, the verifications fail. Hence $\mathsf{Hyb}_{\mathsf{pred}(t)}$ obtains $m = \epsilon$ and sets $out_{i',s',r} = \epsilon$.

- $b = 1$: Then Hyb_t would decrypt c to $m_1 = (s, sig)$ with $sig = \mathsf{sign}_s(in_{i,s,r}, i, r)$. It first tests that $s = s'$. If yes, it verifies that $\mathsf{test}_s(sig) = (m', i' - 1, r)$ for any $m' \in Msg$ and the given (i', r). As above, this implies $i' - 1 = i$; hence the verifications fail and $out_{i',s',r} = \epsilon$.

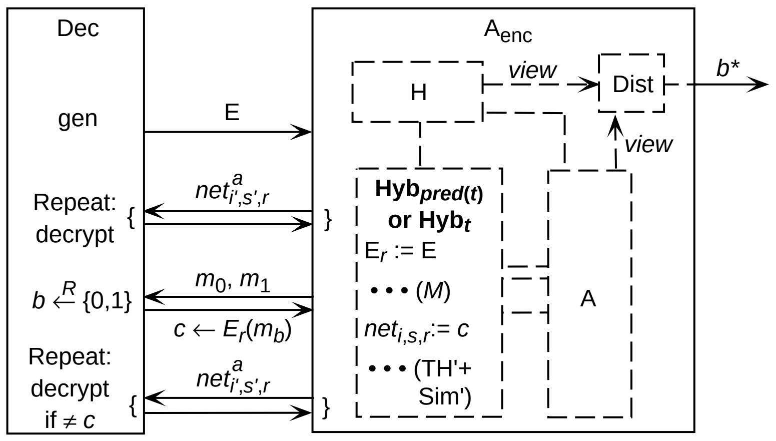

Figure 3.7: Reduction to an adversary on the encryption scheme

Hence the success probability of Dist within $\mathsf{A_{enc}}$ is the same as in its normal setting. Since $\mathsf{A_{enc}}$ outputs the same value b^* as Dist, we can compute the success probability $p_{enc}(k)$ of $\mathsf{A_{enc}}$ as follows:

$$\begin{aligned} p_{enc}(k) &= \frac{1}{2}P(b^* = 1|b = 1) + \frac{1}{2}P(b^* = 0|b = 0) \\ &= \frac{1}{2}p_t(k) + \frac{1}{2}(1 - p_{\mathsf{pred}(t)}(k)) \\ &= \frac{1}{2} + \frac{1}{2}(p_t(k) - p_{\mathsf{pred}(t)}(k)) \\ &> \frac{1}{2} + \frac{1}{2Q(k)} \end{aligned}$$

for infinitely many k. This contradicts the security of the encryption scheme, and thus the assumption that a distinguisher Dist exists as described. Hence $view_{conf_{\mathsf{hyb},0}}(\mathsf{A},\mathsf{H}) \approx view_{conf_{\mathsf{hyb},t_{max}}}(\mathsf{A},\mathsf{H})$. As $\mathsf{Hyb}_0 = \mathsf{TH}' + \mathsf{Sim}'$ and $\mathsf{Hyb}_{t_{max}} = \mathsf{M}'$, this means

$$view_{conf'_2}(\mathsf{A},\mathsf{H}) \approx view_{conf'_1}(\mathsf{A},\mathsf{H}).$$

All parts of the proof together and transitivity therefore imply the desired result

$$view_{conf_1}(\mathsf{A},\mathsf{H}) \approx view_{conf_2}(\mathsf{A},\mathsf{H}).$$

■

4 Formal Verification with CSP

In this chapter, we present first results of work on Objective 2 of this workpackage (cf. Section 1.2), i.e., formal modeling and verification in the process algebra CSP. CSP stands for “Communicating Sequential Processes” (introduced in [32], developed in [53]).

The CSP user community has been modeling security protocols using standard synchrony models almost since the arrival of FDR [21] (the first commercially available analytical CSP model checker) about nine years ago. There have been some notable – and high-profile – successes, and the theory is now mature, with numerous papers and books having been written on the subject. However, we are sure that the scale of the MAFTIA project will provide fresh challenges.

CSP is a process algebra which is useful for describing systems that interact by communication. A system is modeled as a *process* (itself possibly constructed from a collection of processes), which interacts with its environment by means of atomic *events*. (Thus a process in CSP corresponds to a machine or a combined collection of machines in Chapter 2.) Communication is of the handshake type (often called synchronous, but in the MAFTIA context “synchronous” has another meaning): an event takes place precisely when both the process and the environment agree on its occurrence. The syntax of CSP provides a variety of operators for modeling processes, and the associated algebra provides rewrite laws. Primitive operators include process prefix, sequential composition, deterministic and non-deterministic choice, parallelism, hiding, recursion, deadlock and successful termination:

$$\begin{aligned} P \ ::= \ & a \rightarrow P \mid P \,; P \mid P \,\Box\, P \mid P \sqcap P \mid P \parallel P \mid P \,|||\, P \mid \\ & P \setminus b \mid \mu X \bullet F(X) \mid STOP \mid SKIP \end{aligned}$$

The collection of mathematical models and associated semantics that make up CSP facilitate the capture of a wide range of process behaviours. The traces model captures the observable traces of events which a CSP process might exhibit. (I.e., it corresponds to runs and their restrictions in Chapter 2.) The failures model captures not only the traces of a process, but also those events it can refuse to perform after a particular trace. The failures divergences model also captures information about events which a process may diverge on (perform infinitely often) from a particular state.

The theory of refinement in CSP allows correctness conditions to be encoded as refinement checks between processes. In the simplest version, trace refinement, process R is refined by process S (written $R \sqsubseteq S$) iff all traces of S are also traces of R. Refinement is transitive:

$$R \sqsubseteq S \wedge S \sqsubseteq T \Rightarrow R \sqsubseteq T$$

If process R is refined by process S and S is refined by T then R is refined by T. All CSP operators are monotonic with respect to refinement:

$$R \sqsubseteq T \Rightarrow C[R] \Rightarrow C[T]$$

This facilitates compositional development of systems:
Imagine we want to prove:

$$Spec \sqsubseteq C[System].$$

We can break this into two parts: Find a process which does refine the desired specification:

$$Spec \sqsubseteq C[P]$$

and prove that:

$$P \sqsubseteq System.$$

By monotinicity:

$$C[P] \sqsubseteq C[System]$$

and by transitivity:

$$Spec \sqsubseteq C[P] \wedge C[P] \sqsubseteq C[System] \Rightarrow Spec \sqsubseteq C[System].$$

The FDR tool takes a machine-readable dialect of CSP, CSP$_M$, as its input syntax, and can be used to check refinements as well as determinism, deadlock freedom and livelock freedom of processes.

Our approach has been to select a number of MAFTIA protocols, based on particular core MAFTIA concepts, which we then model in CSP and verify using FDR. The aim is to create protocol models which consist of both generic MAFTIA security and system concepts, and protocol-specific behaviour. An example of such a separation can be found in Sections 4.2.4 - 4.2.7 below. Here a generic fully synchronous communications network is developed which incorporates an adversary, also described using a generic technique. This separation enables easy re-use of the generic techniques within other protocol models, however, achieving this separation is non-trivial.

We have sought to identify any idiosyncratic properties which appear commonly exhibited by MAFTIA synchronous reactive protocols, and which one may exploit, in one way or another, when writing the CSP models. Also, we aim to identify any relatively atypical security or timeliness properties that we shall commonly want to prove are held by such protocols, and investigate ways of verifying those properties through CSP refinement checks. In this way, we aim to build up a modest library of generic modeling techniques, and references to modeling techniques, which are applicable to the CSP verification of general systems built on MAFTIA protocols and MAFTIA fault-tolerant concepts.

In the following, we first present an example, contract signing, in its original terms (Section 4.1). The general modeling techniques follow in Section 4.2 with some references to this example. The concrete formalization is introduced in Section 4.3.

4.1 Example Protocol: Optimistic Contract Signing

4.1.1 Introduction to Optimistic Contract Signing

A *contract* is a non-repudiable agreement on a given text [12]. A contract signing scheme includes at least three players and two protocols: Two signatories participate in a contract signing protocol "sign" which fairly computes a contract, i.e., guarantees that either both or none of the signatories obtains a contract. This contract can then be used as input to a contract verification protocol "show" to convince arbitrary verifiers such as a court that the signatories reached agreement on the given text.

Note that unlike the original cryptographic contract signing protocols [12], our notion does not tolerate uncertainty about the outcome. In the end, the user must have a definitive answer whether a valid contract was produced or not. Furthermore, we achieve deterministic fairness if the underlying digital signature scheme is secure.

In all practical schemes, contract signing involves an additional player, called third party. This party is (at least to some extent) trusted to behave correctly, thus playing the role of a notary in paper-based contract signing. A well-known protocol for contract signing by exchanging signatures via a third party works as follows [50]: Both signatories send their signatures to the third party. The third party then verifies and forwards them. At the end, both signatories end up having two signatures on the contract which may be sent to any verifier for verification. In this and similar protocols, the third party has to be involved in *all* executions of the contract signing protocol. This is called an inline third party.

In order to minimize this involvement while guaranteeing fairness, the concept of "optimistic contract signing" has been introduced [5]. The basic idea of optimistic schemes is that the third party is not needed in the fault-less case: After the execution of the optimistic signing protocol, two correct signatories that agree on the contract to be signed always end up with a valid contract. Only if one of the signatories misbehaves, the third party is involved to decide on the validity of the contract.

In the following, the requirements on a contract signing protocol and the actual protocol chosen for CSP validation are presented in the original, cryptographic style. They are essentially from [43, 55].

4.1.2 Notations and Assumptions

We assume that a digital signature scheme $(\mathsf{gen}, \mathsf{sign}_{\mathcal{A}}, \mathrm{test}_{\mathcal{A}})$ for a given message-space M and security parameter n is given [27, 18]. The keys for each party, e.g., A, are computed by the probabilistic algorithm $\mathsf{gen}(\cdot)$. For a given message $msg \in M$, $s := \mathsf{sign}_{\mathcal{A}}(msg)$ denotes the signature of a player named $\mathcal{A}$ under a message msg. Such digital signatures computed by A can be verified using the corresponding verification function $\mathrm{test}_{\mathcal{A}}$, which is distributed using a given certification infrastructure. A signature s on a message m

is called *valid* iff $\text{test}_{\mathcal{A}}(s,m) = \mathsf{true}$. The security of the signature scheme guarantees authentication and non-repudiation, i.e., sign_A can be used to compute a valid signature on m and without knowledge of sign_A, a polynomial-time adversary cannot compute valid signatures s' on messages m' not previously signed by A, except with negligible probability. In our protocols, we assume that $\mathcal{A}$ as well as *msg* can efficiently be computed given the signature. Furthermore, the security analysis of our protocols is done as if digital signatures would provide error-free authentication, i.e., we do not consider the negligible case that signatures may be broken.

With $|S|$ for any set S, we denote the number of the elements in set S.

We assume that signed messages are typed and labeled with the protocol parameters, e.g., that sending $m_2 = \mathsf{sign}_O(\mathit{text})$ in protocol "prot" using a third party T executed by machine S running with a *tid* started at time t_0 (if synchronous) to machine R actually sends the signed message $\mathsf{sign}_O(\text{"}\mathsf{prot}\text{"}, \mathsf{S}, \mathsf{R}, \mathsf{T}, \mathit{tid}, t_0, \text{"}\mathsf{m\_2}\text{"}, \mathit{text})$ in order to prevent interchanging messages between different protocols and runs (the identifier "$\mathsf{m\_2}$" denotes the unique name identifying message m_2; for clarity of our protocols, we may nevertheless mention some of the parameters that are included and signed automatically). Messages without this form or with unexpected parameters are simply ignored. In the synchronous case, messages that do not arrive in their designated round are ignored.

In our figures, $Ⓐ \xrightarrow{a/b} Ⓑ$ depicts that a machine is in state A and receives a message called a. It sends a messages called b and changes to state B. Sending multiple messages $b_1, b_2, \ldots$ is denoted by sending $b_1 + b_2 + \ldots$. The recipient of a message is not depicted. It is described in the text and can be determined by searching for an input of the given message at another machine. Dashed arrows denote non-optimistic exception handling. If the message name is bold, the message is exchanged with the third party. Subscripts in message names usually denote the time at which they are sent (e.g., m_3 would be a message from Round 3). Bold states are final states. If a message m_i is *not* received on a synchronous network, this is modeled by receiving the message $\neg m_i$.

Our figures depict the automata for one run with a given *tid*, only. This run is identified by the *tid* received in the first input or message. To enable parallel execution of multiple protocols, a new process needs to be started for each new *tid* received. A state-keeping third party, for example, will always listen to incoming messages. If it receives a message, it checks whether a process for this *tid* and these signatories exists. If this is the case, the message is forwarded to this process. Else, a new copy of the third party automaton is started in a new process and parameterized with the new *tid*.

4.1.3 Definitions

The following requirements are formulated for both the synchronous and the asynchronous model. We present both because verification of an asynchronous protocol is also intended. On synchronous networks, messages are guaranteed to be delivered within a "round", i.e., a recipient of a message can decide whether a message was sent or not.

On asynchronous networks, messages are eventually delivered but may be delayed and reordered arbitrarily. A machine is called *correct* if it adheres to its algorithm. For incorrect machines, a malicious fault model is made.

4.1.3.1 Optimistic Contract Signing

The following requirements cover different flavors of contract signing. All assume that the signatories agreed on the unique and fresh *tid* and know their mutual identities before starting the protocol. A common method to guarantee this is to use a pair of two locally unique numbers as the global transaction identifier. Note that subsequent protocol executions (e.g., "sign" and "show") are linked using the input *tid*. On synchronous networks, the players must also have agreed on a round t_0 in which the protocol is started.

Definition 4.1 (Contract Signing Scheme) *A* contract signing scheme *for a contract space M with $|M| \geq 2$, an identifier space id_space, and a set of transaction identifiers TIDs is a triple* $(\mathsf{A}, \mathsf{B}, \mathsf{V})$ *of probabilistic interactive algorithms (such as probabilistic interactive Turing Machines) where* V *does not keep state between subsequent protocol runs. The algorithms* A *and* B *are called signatories, and* V *is called verifier. In addition, the scheme may specify a set AM of auxiliary machines without in- or outputs. The algorithms can carry out two interactive protocols:*

Contract Signing (Protocol "sign"): *Each signatory, e.g.,* A*, obtains a local input* $(\mathsf{sign}, \mathcal{B}, C_A, tid)$ *where* sign *indicates that the* "sign"*-protocol shall be executed with signatory $\mathcal{B} \in$ id_space, $C_A \in M$ is the contract text* A *wants to sign, and $tid \in TIDs$ is the common unique transaction identifier. At the end, each of* A *and* B *returns a local output, which is either* (signed, tid) *or* $(\mathsf{rejected}, tid)$*.*

Verification (Protocol "show"): *This is the contract verification protocol between any particular verifier* V *and only one of the signatories* A *or* B*.*[1][2] *The signatory, say* A*, obtains a local input* (show, tid)*. The verifier* V *outputs either* $(\mathsf{signed}, \mathcal{A}, \mathcal{B}, C, tid)$ *with the identities*[3] *of the two signatories and the contract text, or* $(\mathsf{rejected}, tid)$*.*

◇

Intuitively, an output signed of the "sign"-protocol means that the user can now safely act upon the assumption that the input contract has been signed, i.e., that a subsequent

[1]Note that we did not consider verification including A *and* B, even though this may lead to more efficient protocols for some models.

[2]Without this restriction, a signatory would not be enabled to switch off its machine as long as the contract is valid.

[3]It is assumed that the order of the names does not matter. E.g., a requirement that the verifier outputs $(\mathsf{signed}, \mathcal{A}, \mathcal{B}, C, tid)$ is also fulfilled if it actually outputs $(\mathsf{signed}, \mathcal{B}, \mathcal{A}, C, tid)$.

verification will succeed. If the protocol outputs rejected, the user can safely assume that no contract was signed, i.e., the other signatory will not be able to pass verification.

The set AM subsumes all auxiliary machines needed for the correct operation of the scheme. Since they do not make in- or outputs, there is no need to identify them individually. Examples include network machines, certification authorities, or third parties.

The requirements on termination of the protocols depend on the underlying network: On asynchronous networks, nobody can decide whether a message will eventually arrive or not. Thus, with incorrect players, termination cannot be guaranteed without precautions. Therefore the user is allowed to request termination manually: After a local input (wakeup, *tid*), the protocol stops waiting for pending messages and is required to terminate and produce a correct output within a limited time, e.g., by only interacting with a correct third party. Note that this enforced termination may lead to a different (but still fair) outcome as compared to an undisturbed run of the protocol.

Remark 4.1. Most existing protocols only consider one single run of a protocol, i.e., they do not introduce transaction identifiers to distinguish two independent runs. ○

Definition 4.2 (Secure Contract Signing) *A contract signing scheme (Definition 4.1) is called secure if it fulfills the following requirements if the machines in AM are correct*[4]*:*

Requirement 4.2a (Correct Execution): *Consider an execution of* "sign" *by two correct signatories with an input* (sign, $\mathcal{B}$, C_A, *tid*) *to* A *and* (sign, $\mathcal{A}$, C_B, *tid*) *to* B *with a unique and fresh tid* $\in$ *TIDs and* $C_A, C_B \in M$*. If these inputs are made in the same round on synchronous networks and if* wakeup *is not input on asynchronous networks, the* "sign"*-protocol outputs* (signed, *tid*) *iff* $C_A = C_B$ *or else* (rejected, *tid*) *to both signatories.*

Requirement 4.2b (Unforgeability of Contracts): *If a correct signatory, say* A*, did not receive an input* (sign, $\mathcal{B}$, C, *tid*) *so far, any correct verifier* V *will not output* (signed, $\mathcal{A}$, $\mathcal{B}$, C, *tid*)*.*

Requirement 4.2c (Verifiability of Valid Contracts): *If a correct signatory, say* A*, outputs* (signed, *tid*) *on input* (sign, $\mathcal{B}$, C, *tid*) *and later executes* "show" *on input* (show, *tid*) *with a correct verifier* V*, then* V *will output* (signed, $\mathcal{A}$, $\mathcal{B}$, C, *tid*)*.*

Requirement 4.2d (No Surprises with Invalid Contracts): *If a correct signatory, say* A*, outputs* (rejected, *tid*) *on input* (sign, $\mathcal{B}$, C, *tid*)*, then no correct verifier will output* (signed, $\mathcal{A}$, $\mathcal{B}$, C, *tid*)*.*

Furthermore, one of the following requirements must be fulfilled:

Requirement 4.2e (Termination on Synchronous Network): *On input of* (sign, $\mathcal{B}$, C, *tid*)*, a correct signatory, say* A*, will either output* (signed, *tid*) *or* (rejected, *tid*) *after a fixed number of rounds.*

[4]Note that a specific system may weaken this strong trust assumption.

Requirement 4.2f (Termination on Asynchronous Network): *On input of* (sign, $\mathcal{B}$, C, tid) *and* (wakeup, tid)*, a correct signatory, say* A*, will either output* (signed, tid) *or* (rejected, tid) *after a fixed time.*[5]

◇

Remark 4.2. Note that wakeup can only be input to the signatories during the "sign" protocol. Termination of the "show" protocol is implied by [R. 4.2c] if a contract was produced.[6]

Remark 4.3. If two correct signatories input different contracts $C_A \neq C_B$ on asynchronous networks and later input wakeup, then an output (rejected, tid) to both is not required by "correct execution". However, if the protocol would not output (rejected, tid) to both, then at least one signatory would obtain an output signed. For "verifiability", this signatory would be able to convince the verifier. This, however, contradicts "unforgeability" of the other correct signatory who did not input the same contract. ○

Definition 4.3 (Optimistic Contract Signing with Two-Party Verification) *An* optimistic contract signing scheme *for a contract space M, an identifier space id_space, and a set of transaction identifiers TIDs is a triple* $(\mathsf{A}, \mathsf{B}, \mathsf{V})$ *together with a machine* T*. The scheme must fulfill the following requirements:*

Requirement 4.3a (Security): $(\mathsf{A}, \mathsf{B}, \mathsf{V})$ *with the set $AM := \{\mathsf{T}\}$ is a secure contract signing scheme (Def. 4.2) where the third party does not participate in the* "show" *protocol.*

Requirement 4.3b (Limited Trust in T): *R. 4.2b and R. 4.2c are fulfilled even if* T *is incorrect.*[7]

Furthermore, one of the following requirements must be fulfilled:

Requirement 4.3c (Optimistic on Agreement on Synchronous Network): *If two correct signatories input* (sign, $\mathcal{B}$, C, tid) *and* (sign, $\mathcal{A}$, C, tid) *in a given round with a fresh and unique tid and a $C \in M$, the third party does not send or receive messages in the* "sign"*-protocol.*

Requirement 4.3d (Optimistic on Agreement on Asynchronous Network): *If two correct signatories input* (sign, $\mathcal{B}$, C, tid) *and* (sign, $\mathcal{A}$, C, tid) *with a fresh and unique tid and a $C \in M$ and do not input* (wakeup, tid)*, then the* "sign"*-protocol terminates in a fixed time and the third party does not send or receive messages.*

[5]This is a logical time: It means that there exists a fixed bound on the maximum number of message exchanges after wakeup. Nevertheless, the notion is stronger than "eventually".

[6]Implicitly, this assumes that a user only executes the "show"-protocol after obtaining a contract.

[7]Intuitively, R. 4.2d cannot be guaranteed for an incorrect T. Since the outcome of one correct signatory must depend on the behavior of T (otherwise, T would not be needed), T could trick this signatory into an output rejected while enabling an output signed to its peer.

An optimistic contract signing scheme is called optimistic on disagreement *if R. 4.3c or R. 4.3d hold even if the correct signatories input different contract texts $C_A \neq C_B$.* ◇

Remark 4.4. R. 4.2a for an incorrect T automatically holds in all optimistic executions (if T is not asked, it cannot influence the outcome). This means that if the participants are correct, agree, and do not input wakeup, all optimistic protocols will automatically produce a correct output signed. Only protocols that are optimistic on disagreement usually produce a correct output of rejected without contacting the third party. ○

A weaker definition allows the third party to participate in the verification of a contract. This enables, e.g., revocation of contracts during recovery:

Definition 4.4 (Optimistic Contract Signing with Three-party Verification) *An* optimistic contract signing scheme with three-party verification *for a contract space M, an identifier space id_space, and a set of transaction identifiers TIDs is a triple* $(\mathsf{A}, \mathsf{B}, \mathsf{V})$ *together with a machine* T *of probabilistic interactive algorithms. The scheme must fulfill the following requirements:*

Requirement 4.4a (Security): $(\mathsf{A}, \mathsf{B}, \mathsf{V})$ *with with a set $AM := \{\mathsf{T}\}$ with a correct machine* T *is a secure contract signing scheme (Def. 4.2).*

Requirement 4.4b (Limited Trust in T): *R. 4.2b is fulfilled even if* T *is incorrect.*

The scheme is called optimistic on agreement if R. 4.3c or R. 4.3d hold. It is called optimistic on disagreement if these requirements hold even if the correct signatories input different contract texts $C_A \neq C_B$. ◇

Remark 4.5. Note that the involvement of the third party in the verification is not as bad as its involvement in the actual contract signing protocol: In practice, only few signed contracts should be disputed at court. ○

4.1.4 A Message-optimal Synchronous Scheme

We now describe the synchronous contract signing protocol which was chosen for formal evaluation. It needs three messages in the optimistic case and uses a state-less third party. Its optimistic behavior is depicted in Figure 4.1. The individual machines of the players are depicted in Figures 4.2, 4.3, and 4.4. This can be shown to be message-optimal [43].

Scheme 4.1 (Message-optimal Synchronous) *This scheme consists of the triple* (A, B, V) *and* T *of interactive probabilistic machines which are able to execute the protocols defined as follows:*

Contract Signing (Protocol "sign"; Figure 4.1): *On input* $(\mathsf{sign}, \mathcal{B}, C_A, tid)$*, the signatory* A *initiates the protocol by sending the signed message* $m_1 := \mathsf{sign}_A(\mathcal{A}, \mathcal{B}, \mathcal{T}, t_0, C_A, tid)$ *with contract* C_A *to the responding signatory* B. B *receives the input* $(\mathsf{sign}, \mathcal{A}, C_B, tid)$ *and message* m_1 *and verifies whether the received contract text* C_A *is identical to the input contract text* C_B*. If not, the players disagree about the contract and* B *returns* $(\mathsf{rejected}, tid)$*. Else, it signs the received message and sends it as* $m_2 := \mathsf{sign}_B(m_1)$ *to* A*. Player* A *then signs the received message again, sends it back as* $m_3 := \mathsf{sign}_A(m_2)$ *and outputs* (signed, tid)*. On receipt of message* m_3*,* B *outputs* (signed, tid) *as well. After a successful execution of this optimistic protocol,* A *and* B *store* m_3 *under the input tid for later use in a verification protocol.*

If A *does not receive message* m_2 *it waits until Round 5, and, if* m_5 *is not received, it outputs* $(\mathsf{rejected}, tid)$*. If* B *did not receive message* m_3*, it may be that* A *nevertheless was able to compute a valid contract* m_3 *after receiving* m_2*. Therefore it starts the* "resolve"*-protocol to invoke the third party to guarantee fairness.*

Recovery from Exceptions (Sub-protocol "resolve"): B *sends a message* $m_4 := \mathsf{sign}_B(m_2)$ *containing* m_1 *and* m_2 *to the third party* T*. The third party then checks whether both players have agreed and then forwards* m_2 *in* $m_5 := m_2$ *to* A*, which might still wait for it. This guarantees that* A *receives a valid contract* $m_3 := \mathsf{sign}_A(m_2)$ *and outputs* (signed, tid)*. Furthermore* T *sends an affidavit on* m_2 *in* $m'_5 := \mathsf{sign}_T(m_2)$ *to* B *and* B *outputs* (signed, tid)*. After the* "resolve"*-protocol,* A *keeps* m_3 *and* B *keeps* m'_5 *to be used in later verification protocol executions.*

Verification of a Contract (Protocol "show"): *On input* (show, tid)*, a signatory looks up* m_3 *or* m'_5 *and sends it to the verifier. The verifier verifies it and outputs* $(\mathsf{signed}, \mathcal{A}, \mathcal{B}, C, tid)$ *if this succeeds and* $(\mathsf{rejected}, tid)$ *else.*

◇

Remark 4.6. For simplicity, our protocols do not keep the contract confidential against the third party. This can easily be improved by signing a contract on $\mathcal{H}(C)$ and modifying the verification as follows: For verifying a contract, a signatory sends C to the verifier and inputs (show, tid) to the contract signing scheme. The verifier then outputs $(\mathsf{signed}, \mathcal{A}, \mathcal{B}, C, tid)$ if the verification outputs $(\mathsf{signed}, \mathcal{A}, \mathcal{B}, h_C, tid)$ and $h_C = \mathcal{H}(C)$ for the received C. ∘

In the following, the hand-made proof from [43, 55] is included for comparison with the later formal methods.

Lemma 4.1 (Security of Scheme 4.1) *Scheme 4.1 is a secure synchronous optimistic contract signing scheme, which is optimistic on agreement.* □

Proof. The scheme adheres to Definition 4.1 by construction. We now show that each of the requirements described in Definitions 4.2 and 4.3 are fulfilled:

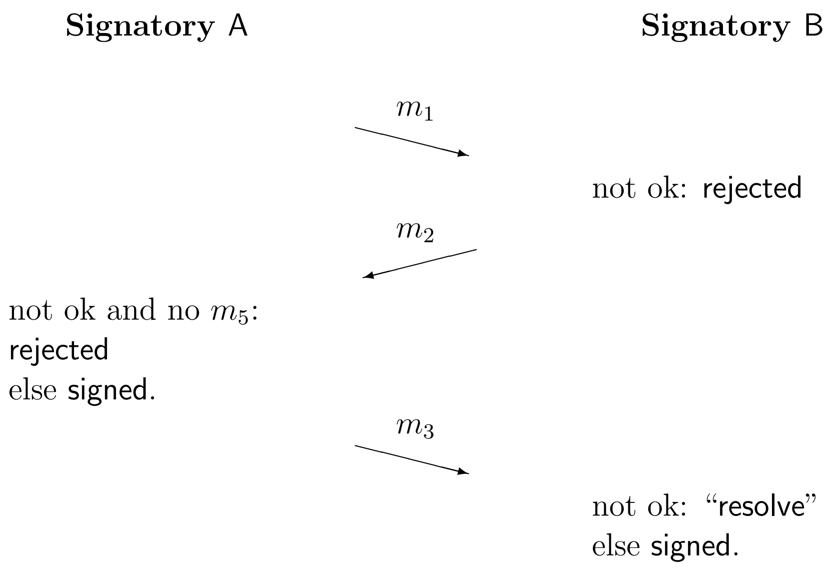
Signatory A
Signatory B
m_1
not ok: rejected
m_2
not ok and no m_5: rejected else signed.
m_3
not ok: "resolve" else signed.

Figure 4.1: Optimistic Behavior of Scheme 4.1.

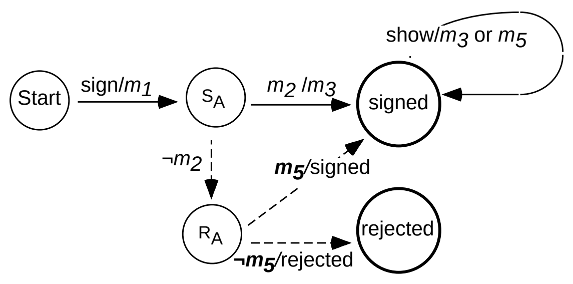

Figure 4.2: Signatory A of Scheme 4.1.

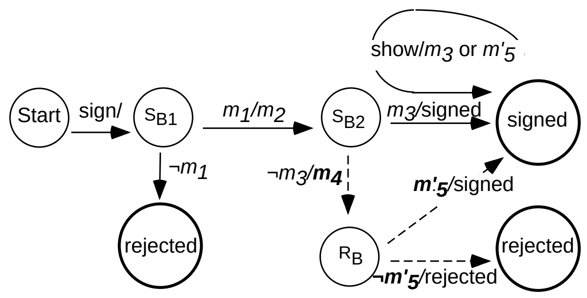

Figure 4.3: Signatory B of Scheme 4.1.

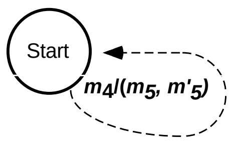

Figure 4.4: Third Party T of Scheme 4.1.

Correct Execution: If both correct players A and B input (sign, $\mathcal{B}$, C_A, *tid*) and (sign, $\mathcal{A}$, C_B, *tid*) with identical *tid* s and $C_A = C_B$, then both receive a valid contract m_3 and output (signed, *tid*). If the contracts or *tid*'s differ, B outputs (rejected, tid_B) after receiving m_1 and A outputs (rejected, tid_A) after not receiving m_5 in Round 5.

Unforgeability of Contracts: In order to convince a correct verifier V for a given *tid*, C, and partner, one needs correct messages m_3 or m'_5 containing this *tid*. Since m_3 as well as m'_5 contain signatures from both participants, a correct signatory input (sign, $\mathcal{A}$, C, *tid*) or (sign, $\mathcal{B}$, C, *tid*), respectively.

Verifiability of Valid Contracts: If a correct machine A outputs (signed, *tid*) then it received m_2 (or m_5 containing m_2) which will be accepted by the verifier as a correct contract m_3 after being signed by A. B outputs (signed, *tid*) only if it received m_3 or m'_5 which are accepted by the verifier, too.

No Surprises with Invalid Contracts: Let us first assume that a correct signatory A returned (rejected, *tid*) on input (sign, $\mathcal{B}$, C, *tid*) whereas B is able to convince the verifier. This requires that B knows m_3 or m'_5 for the given *tid* and C. Since A returned rejected, it did not receive m_2 until Round 5 and it did not send m_3. Therefore, only m'_5 could lead to successful verification. However, if the third party was correct, it will not accept recovery requests from B after Round 4[8]. Furthermore, in Round 4, no recovery was started since A did not receive m_5 in Round 5. Thus B did not receive m'_5 in Round 5.

Let us now assume that a correct signatory B returned (rejected, *tid*) on input (sign, $\mathcal{A}$, C, *tid*). Then B either did not send m_2 for this *tid* or it sent m_4 to T. In the first case, A cannot convince the verifier since m_2 is part of m_3 and also m'_5. In the second case, a correct third party would necessarily have answered with m'_5 and thus B would not have returned (rejected, *tid*).

Termination on Synchronous Network: The scheme requires at most 5 rounds (3 in "sign" and 2 in "resolve") to terminate.

Limited Trust in T: Even if T is incorrect, it cannot forge any signature of the two signatories. Therefore, unforgeability still holds.

[8]This round number is relative to the time t_0 as fixed by the initial input of A and implicitly included and signed in m_1 (see Section 4.1.2).

Verification is independent of T. Therefore, verifiability even holds if T is incorrect.

Optimistic on Disagreement on Synchronous Network: If the correct signatories input (sign, $\mathcal{B}$, C_A, *tid*) and (sign, $\mathcal{A}$, C_B, *tid*) with $C_A = C_B$, signatory A outputs (signed, *tid*) after Round 2 whereas player B outputs (signed, *tid*) after Round 3. In this case, the scheme sends the m_1, m_2, m_3 and requires 3 rounds.

If $C_A \neq C_B$, A outputs (rejected, *tid*) after Round 5 and B after Round 1 without contacting the third party by starting "resolve". In this case, the scheme sends m_1 only but requires 5 rounds.

■

4.2 The General CSP Modeling Technique

In this section we describe the construction of a CSP model, *SYNC*, of a synchronous communication network. We then describe how protocol-specific adversaries, or 'spies', are incorporated into the *SYNC* model. The adversary needs protocol-specific parts, in contrast to Chapter 2, because, as in all "classical" formal security models, abstractions from cryptography are built into it. We shall refer to any model that results from incorporating some protocol-specific spy into the communication network model as a *SYNC_WITH_SPY* model. A *SYNC_WITH_SPY* model was developed as part of the formal verification of the synchronous contract-signing protocol, and this model is discussed in Section 4.3 below. Compared with Chapter 2, it corresponds roughly to a model of runs with a particular clocking scheme (a rushing adversary), a framework for the adversary's machine, and insecure channels. Secure channels, authenticated channels, and fail/stops (as opposed to just failure) can easily be added to the synchrony model as presented here. We will discuss how the model may be further developed to accommodate those refinements at the appropriate point.

At first sight the CSP model may appear to differ to the formal model, in that the spy may *attempt* to deliver non-deterministically any number of inferred messages in the same sub-round. However, if a protocol process is written according to the formal model, then this makes no difference, as that process will only ever be prepared to receive at most one message from each party in any particular round, and the CSP algebra will oblige the spy process to adhere to that constraint.

Our main concern in developing the *SYNC* model is that it should be practical and reusable, and as generic as possible. Further refinements – for example secure channels and complete stopping of processes – can be easily included if need be. The *SYNC* model is, therefore, a good base model for the purposes of the verification of distributed synchronous MAFTIA protocols. After incorporating the protocol-specified spy, the distributed components will communicate 'over the network' using the appropriate channels, as specified by *SYNC*.

By way of example, the complete model of our synchronous contract-signing session will be $SYNC\_WITH\_SPY$ communicating with a trusted third party (TTP), a 'Verifier machine' ($VERIFIER$), plus a number of instances of the contract-signing protocol, ($SIGN$), parametrized by the identifiers of signatories (excluding the 'corrupted' parties that are wholly incorporated in $SYNC\_WITH\_SPY$). This will look something like:

$$
\begin{array}{l}
SYNC\_WITH\_SPY \\
[|send, receive, tock|] \\
(\ (\ TTP\ [|tock|]\ VERIFIER\ ) \\
[|tock|] \\
(\ [|tock|]\ p : P - CORRUPT\ @\ SIGN(p, \ldots)\ )\ )
\end{array}
$$

The process of modelling $SYNC\_WITH\_SPY$ is as self-contained and general as possible. The only protocol-specific references are to the datatype, Msg, of the messages that are to be conveyed by $SYNC$'s communication channels, and to a set of rules that specify the adversary's capabilities - i.e., what 'secret' messages it may 'overhear', and how, if at all, it is able to 'forge' messages from those that it has already 'overheard'. This set of rules is specified by a *Deductions* function. Msg and *Deductions* are the protocol-specific 'ingredients' necessary to model a $SYNC\_WITH\_SPY$.

Up to the point where the *Deductions* rules are incorporated into the network communication model, $SYNC$, we can talk in general terms, i.e., without reference to protocol specifics. However, incorporating any particular spy's inference rules into the communication model in such a way that the $SYNC\_WITH\_SPY$ model will be practical, is a non-trivial task in general. At this point, we invariably have to 'look inside' Msg and *Deductions* for protocol-specific idiosyncrasies that we might exploit in various modelling optimizations.

The $SYNC$ model was designed to afford as complete as possible coverage of security properties through FDR refinement checks. It was possible to heavily optimize the contract-signing $SYNC\_WITH\_SPY$, and its asynchronous variant.

The models contained at most four distributed parties: two signatories, the TTP and the Verifier machine. These parties communicated some ten different messages over the network during the course of a contract-signing session. However, in general, parameter sizes (number of parties, messages etc.) will vary. We would expect more than six medium sized principals running over a network to yield models intractable for FDR. This is discussed further in Sections 4.4.3 to 4.2.7.

The $SYNC$ CSP$_M$ code presented in this section relates to a synchronous communication network with fixed link latency and a fully-connected, invariable topology. Future MAFTIA networks will require more complex topologies. In Section 4.4.2, we describe how our basic $SYNC$ model may be modified so as to admit different link latencies (hybrid timing models) and more complex link topologies.

At appropriate points we shall remark on various modeling methodologies that can be used to simplify the coding-up of various network components – e.g., how to avoid unintentional 'time-stop' deadlocks. In Section 4.4.4, we discuss how 'liveness properties' relevant to our MAFTIA models, such as 'eventual termination', can be specified in CSP.

4.2.1 Datatypes

Throughout this section, P will denote the CSP datatype representing the identities of certain distributed parties communicating over the network. The tag F, $F \in P$, is reserved for the identity of an adversary or 'spy'[9]. $CORRUPT$ is a fixed subset of P, containing F, which represents a number of 'corrupted' nodes ($P - CORRUPT$ are the 'correct' parties). Msg is the datatype representing the payload type of messages sent over the network. (Thus, compared with Chapter 2, a specification is for a specific structure, and $P - CORRUPT$ corresponds to a set $\mathcal{H}$ in an access structure.)

We shall refer to other, protocol-specific, datatypes, such as the transaction identifiers, TID, of the contract-signing protocols. These datatypes will be introduced as and when appropriate.

4.2.2 Modeling Time and the *tock* Event

It is to be expected that timing and synchrony assumptions will be stated in slightly different ways, and with varying degrees of formality by different protocol authors. In [55], for example, the formal Lynch synchrony and timing model is assumed for the Contract Signing protocols[10]. In the fully synchronous theoretical model, which we are assuming here, our communication medium will deliver all messages within a known bounded time. There are known bounds on processing delays and clock drifts.

We model time in CSP$_M$ using a discrete approximation of real-time afforded by the proverbial *tock* event. The *tock* event, in general usage, can be interpreted in several subtly different ways as appropriate to the modeling of the protocols in question. Probably the most familiar interpretation of *tock* is as a reliable indicator of time passing – i.e., a fixed latency period having 'elapsed' between any two consecutive *tock* events. Another interpretation of *tock*, is as a global 'end-of-round' marker, and this is how it should be read here. All processes that depend on timing constraints, and have the same notion of time, must synchronise on the global *tock* event.

It is possible that any time-dependent process will retain, as part of its state, some parameter recording the passing of time, implemented by counting how many *tock* events have passed since some local 'time zero' (which may or may not be reset). In the synchronous contract-signing protocol, for example, a correct signatory process needs to retain a parameter, r, for every session that it is currently engaged in. Each such r records the number

[9]We chose F to stand for 'faulty' (as opposed to strictly malicious) behaviour. Other options for the adversary's tag, e.g. 'S' for spy or 'A' for adversary, would have conflicted with the naming convention used in the synchronous contract-signing protocol model.

[10]The Contract Signing protocols are unusually well described for cryptographic protocols. Even so, the timing and synchrony assumptions were still read slightly differently by two teams of modelers.

of *tock*s that have passed since a session was initiated – r is the 'round' that the session is currently in.[11]

We can only model check finite-state processes, so an upper bound must be imposed on our clocks, and this may constrain the completeness of our checks. However, it is often the case that when time-dependent information reaches a certain age it is no longer important how old it is. Therefore, we are able to put a limit on the age of information which might influence the behaviour of the system, we shall call this 'retirement'. We can exploit this by, for example, only counting time on this information until the point of retirement, or modeling the timing of our protocol processes modulo the retirement period. The period of retirement invariably has to be defended by informal argument, this is standard practice in finite-state model checking.

A single dedicated global 'clock' process may be used to count the *tock*s. All time-dependent processes must still agree on the *tock* event, but they are no longer obliged to keep individual state parameters recording time. When one of the distributed processes wants to know what the current time or round number is, it may ask for that information from the clock process. This could be implemented by running the clock process in parallel and communicating over a dedicated channel, *get_time*, say. The clock process increments an internal counter, t, whenever a *tock* occurs, and is always prepared to engage in $get\_time!t$. Any process engaging in $get\_time?t'$ will then have the current time returned in its local variable t'. This modeling technique can sometimes be used to save on state-space when there are a large number of time-dependent distributed processes running over the network.

4.2.3 Communication Channels

Recall that in CSP, an event takes place precisely when both the process and the environment agree on its occurrence. If one immediately uses such events to transport messages, every message would arrive at once. Hence we have to explicitly model the network that may buffer messages for one round.

The model of the communication network employs two channels, *send* and *receive*. The *send* channel carries three objects: *sender*, *receiver* and *payload*. The event $send.A.B.m$ represents node A sending a message with payload m to node B, $A \neq B$. The *receive* channel also carries three objects: *receiver*, *sender* and *payload*. The event $receive.B.A.m$ represents the receipt of a message with payload m sent by node A to node B. The *sender* field of the *receive* channel is necessary so that protocol processes do not have to

[11] Although in the generalized version of the synchronous contract-signing model presented in 4.3, we allowed the protocol to respond early to early receipt of messages. That is, we modelled the protocol so that it did not necessarily wait to the next round to respond to the receipt of a message, and so there is not a direct correlation between 'number of tocks elapsed' and 'round'. Nevertheless, the *tock* counter, r, is still necessary in order that the protocol can determine whether or not a message would ever be delivered. The TTP also needs to keep a global counter, so that it can ascertain whether or not a message sent to it as part of the 'resolve' process is out-of-date with respect to the time at which the session was began.

synchronise on the sends of other protocol processes. The *receiver* field is necessary in order that our synchronous network process be able to guarantee delivery of a message, within a round, to the intended recipient.

It is important that the processes which communicate with the network only contain events in their alphabets which represent communications either to or from themselves. That is, if A is a correct (or reliable) node process with alphabet $\alpha(A)$, say, then it should be the case that $\alpha(A) \cap (\{|send|\} \cup \{|receive|\}) = \{|send.A|\} \cup \{|receive.A|\}$. (The notation $\{|x|\}$ means all events starting with x.) If this is not the case, then it may be possible for processes to cause unintentional deadlock conditions. In a timing context these are sometimes called 'time-stops'. For example, if process A synchronises on *send.B* events, then its cooperation is required whenever B wishes to send to any other party. Thus, it is possible for A to deadlock process B, and perhaps the whole model.

4.2.4 A Basic Synchrony Model

We assume, for the time being, that all links of our basic communication network have the same *latency* bound, the time period between any two consecutive *tock*s. If all protocols are modeled strictly according to the formal model, and all links have the same *latency*, then the *tock*s will correspond directly to end-of-round, and a *latency* of 1 is sufficient. This will certainly be the norm for MAFTIA verifications. In the CSP$_M$ code that follows, however, we shall refer to an integer *latency* constant so that we may later illustrate how our model might be modified to account for differing latency periods, should that ever be necessary.

We first fix two correct communicating parties, identified by A and B, and a message $m_0 \in Msg$. Let $SYNC_{A,B,m_0}$ be the process that models A communicating m_0 to B over the network. A naïve first model of network communications would then be the $SYNC_{A,B,m_0}$ processes in parallel, synchronising on *tock*:

$SYNC = [|tock|]\, A : P,\, B : P - \{A\},\, m_0 : Msg\ @\ SYNC_{A,B,m_0}(false, 0, false)$

The $SYNC_{A,B,m_0}$ process itself is quite straightforward. It begins by only engaging in tocks until, if ever, the node A process wants to engage in $send.A.B.m_0$. After engaging in the $send.A.B.m_0$ event, the message m_0 would be buffered for up to $latency - 1$ *tock*, after which the message would be 'forwarded' to recipient B via the $receive.B.A.m_0$ event. *tock*s can happen at any time subject to the proviso that at most $latency - 1$ *tock*s occur between a $send.A.B.m_0$ event and the corresponding $receive.B.A.m_0$ event. Note that $SYNC_{A,B,m_0}$ allows to send m_0 repeatedly, but will not guarantee the delivery of a copy of m_0 if a copy of m_0 was sent previously but has not yet been delivered, see Section 4.2.5.

$SYNC_{A,B,m_0}$ can be coded in CSP_M as in Figure 4.5

$SYNC_{A,B,m_0}(know, n, received) =$

know is a boolean indicating whether or not the message has been sent by A. n records the number of *tocks* that have elapsed since the message was sent. *received* is a boolean indicating whether or not the message has yet been delivered to B.

```
( $send.A.B.m_0 \rightarrow$
 $if\, know == false\, then$
        $SYNC_{A,B,m_0}(true, 0, false)$
 $else\, if\; know == true\; and\; received == false\; then$
        $SYNC_{A,B,m_0}(true, n, false)$ )
 $else$
        $SYNC_{A,B,m_0}(true, 0, false)$ )
[]
$know == true\, and\, received == false$ & (
        $receive!B!A!m_0 \rightarrow SYNC_{A,B,m_0}(know, n, true)$ )
[]
$(know == false\, and\, received == false)$ $or$ $(know == true\, and\, received == true)$ & (
        $tock \rightarrow SYNC_{A,B,m_0}(know, n, received)$ )
[]
$know == true\, and\, (received == false\, and\, n < latency - 1)$ & (
        $tock \rightarrow SYNC_{A,B,m_0}(true, n + 1, received)$ )
```

Figure 4.5: The CSP$_M$ processing of a message m_0 sent by correct A to B

4.2.5 Same-Round Multiple-Sends

The basic $SYNC$ process as described in Section 4.2.4 cannot cope with same-round multiple-sends. That is, if node A had wanted to send two or more copies of the same message, m, to the same node B in the space of one round, then the $SYNC$ process could not *guarantee* to deliver more than one copy of m to B by the next round – some copies of m may be 'lost'.

However, we suppose that this is unlikely to be a big handicap in practice – as it is improbable that any synchronous protocol's correctness will be dependent on re-sending some message m to the same recipient before it can even be sure that the recipient had received the first copy of m[12]. This is a behaviour we might expect to see exhibited by an asynchronous protocol, but in this case same-round multiple sends are not an issue because whether or not any particular message is delivered 'on time' is, anyway, captured non-deterministically in CSP.

[12]The synchronous contract-signing protocol does not. In fact, we can exploit the fact that the synchronous contract-signing protocol never requires a correct machine to send any message m to the same recipient twice - the contract-signing $SYNC$ only guarantees forwarding m to the intended recipient the first time it is sent.

If same-round multiple sends are really necessary in order for a protocol to function correctly, then the $SYNC_{A,B,m_0}$ process of Figure 4.5 can be modified to provide for a partial solution.[13]

4.2.6 Integrating the Adversary and Faulty Behaviour into the Network

In automated verification, we seldom make any distinction between ‘intelligent’ and ‘unintelligent’ malicious behaviour, or even between intentionally malicious and faulty behaviour. The adversary must be modeled so that it is free to carry out ‘all possible’ attacks admitted by the protocols, and exhibit ‘all possible’ faulty behavior.

A node is either correct, or it is corrupt. In the latter case, instead of the corrupt node being represented by a separate process in our integrated model, its behavior is fully incorporated into the network communications plus adversary. We call this process $SYNC\_WITH\_SPY$. As all communications in this environment are message-based, ‘attacks’ are dictated by the set of messages that the adversary has possibly ‘overheard’ or ‘learnt’ up to now, and the limitations on the adversary’s ability to ‘infer’ facts from those messages. [14]

We also assume that channels are insecure but reliable, i.e., as with the insecure channels in Section 2.12, the spy can read all messages and insert additional messages, but messages between honest participants do arrive unmodified.

Next we must specify the spy’s ‘rules of deductibility’ – i.e., what messages can the spy infer from the set of messages which it has overheard (or ‘learnt’) up to now? Given such a set of rules, our model of the spy need only record what messages have been learnt, in order for it to be able to play-out all possible next ‘attacks’ during the course of an exhaustive FDR refinement check. Our deduction rules will be defined by the $Deductions$ function, which is a function $\mathbb{P}(Msg) \times \mathbb{P}(Msg) \rightarrow \mathbb{P}(Msg)$. For example, $Deductions(\{\}, CORRUPT)$ is the adversary’s ‘initial knowledge’, and $Deductions(\{m\}, \{F, A\})$ is the set of messages the adversary could deduce from the single message $\{m\}$, given that node A had been corrupted (so that all A’s secrets were known to the spy). (As we shall see later, sometimes it is useful to exclude the spy’s initial knowledge from $Deductions(\{m\}, CORRUPT)$.)

[13] A general solution to the problem requires an infinite number of states. One partial solution entails our keeping a count of how many copies of m were sent and delivered in each round. The total number of undelivered ms at the end of round $r + 1$ must never exceed the number of ms that were sent in round r. This modification would be costly in terms of extra state, and, obviously, is imperfect as one still has to impose an upper bound on the counts. Another partial solution, probably more efficient, but more messy, is to interleave a number of identical processes each processing a single send of m from A to B. One such process would be required for every time A needed to send m to B.

[14] The ‘limitations’ invariably arise from the protocol’s usage of particular cryptographic primitives, such as an encryption or signature scheme, and on the fact that ‘arbitrary polynomial-time behaviour’ cannot be expressed in CSP, in contrast to Chapter 2. They are a CSP expression of the abstractions to be made faithful in Objective 3 of this workpackage.

Although it would be quite straightforward to code a $SYNC\_WITH\_SPY$ model that referred to an entirely general *Deductions* function, this would produce a large possibly intractable model. The CSP modelling that follows is a special case, in which we incorporate the contract-signing *Deductions* into $SYNC$.

In the synchronous contract-signing model, the recursively defined Msg datatype represented the set of *all* messages, 'signed' and 'unsigned' that could be construed as being at all pertinent to a contract-signing session. However, as the CSP$_M$ world is finite-state, we must restrict our 'universal' message-space to a finite set. For the synchronous contract signing model, this was the $msg3$ subset of Msg, where $msg3$ is the set of all messages signed up to three times. ($msg2$ and $msg1$ are the set of all messages signed up to two times and once respectively.) $msg3$, therefore, is our 'payload' datatype.

All $msg3$ elements are of the form $S.a_1.S.a_2.....S.a_n.U.a.b.r.c.t$ where: $n \leq 3$, the as and bs are identifiers of P pertinent to the session in question; r is an integer 'initial round number'; c is a 'contract'; t is a 'transaction identifier'. An example of a $msg3$ element is $S.TTP.S.B.S.A.U.A.B.r_0.c.t$. The 'unsigned part' of this message is $U.A.B.r_0.c.t$. (Used to convey the details of a contract-signing session with transaction identifier t. A and B are the signatories, c is the contract to be signed, and r_0 is the round in which the users initiated the session.) If $p \in P$, then the string $S.p$ prefixed to a message is read as party p having 'signed' that message.

The deduction rules for the contract-signing protocol are relatively straightforward. If message m had been overhead by the adversary, then the adversary may infer from that message alone, any message m' of $msg3$ which is such that if all leading 'signatures' of corrupted parties are deleted from m', then the resultant string is a suffix of m. This basically translates as the spy not being able to 'insert' the signature of a corrupted party before the signature of a correct party. Any other malicious manipulation of a message by the spy would be detectable by an honest party, who could then discard the message. This assumption is justified in more detail in Section 4.3.2. The adversary's initial knowledge is of all messages that are unsigned, or signed only by $CORRUPT$ parties.

In the contract-signing all operators are unary, and thus the following property is true: if $m_1, m_2 \in Msg$ are any two messages, then the adversary could not deduce anything more from $\{m_1, m_2\}$, than what it could deduce from $\{m1\}$ plus what it could deduce from $\{m_2\}$. This property is very useful, because it means that we can code $SYNC\_WITH\_SPY$, as below, easily and very efficiently as a number of $SYNC_{A,B,m_0}$ processes in shared parallel. The $SYNC_{A,B,m_0}$ processes do not need to share state, each of the $SYNC_{A,B,m_0}$ processes referring to a single message only.[15]

This type of model construction can deliver very significant compilation and state-space savings. FDR is good at compiling small processes put in parallel, especially if those processes synchronise on a small set of events – the nearer to parallelism the better (and

[15]Such a property will certainly not always hold – perhaps the simplest counter-example being the case when a Msg datatype is closed under concatenation (at least up to some upper bound on the length of messages). This raises the question of 'strong-typing', which we discuss in more detail in Section 4.4.5.

in $SYNC\_WITH\_SPY$'s case the processes synchronise on *tock* only).

Our contract-signing $SYNC\_WITH\_SPY_{A,B,m_0}$ process is as in Figure 4.6.

```
$SYNC\_WITH\_SPY_{A,B,m_0}(know, n, received) =$
    $( send.A.B.m_0 \rightarrow$
     $if\ member(B, CORRUPT)\ then$
        $SYNC\_WITH\_SPY_{A,B,m_0}(true, 0, true)$
     $else$
        $SYNC\_WITH\_SPY_{A,B,m_0}(true, n, received)\ )$
    $[]$
    $know == true\ and\ received == false\ \&\ ($
        $receive!B!A!m_0 \rightarrow SYNC\_WITH\_SPY_{A,B,m_0}(know, 0, true)\ )$
    $[]$
    $know == false\ or\ received == true\ \&\ ($
        $tock \rightarrow SYNC\_WITH\_SPY_{A,B,m_0}(know, 0, received)\ )$
    $[]$
    $know == true\ and\ (received == false\ and\ n < latency - 1)\ \&\ ($
        $tock \rightarrow SYNC\_WITH\_SPY_{A,B,m_0}(true, n + 1, received)\ )$
    $[]$
    $know == true\ \&\ ($
        $receive?r : P - CORRUPT.F?m : Deductions(\{m_0\}, CORRUPT) \rightarrow$
        $if\ r == B\ and\ m == m_0\ then$
            $SYNC\_WITH\_SPY_{A,B,m_0}(know, n, true)$
        $else$
            $SYNC\_WITH\_SPY_{A,B,m_0}(know, n, received)\ )$
```

Figure 4.6: The CSP$_M$ processing of a message m_0 sent by correct A to B in the presence of the spy

That is, any time after m_0 is first overheard, any message that can be deduced from m_0 alone may be sent to any of the correct parties.[16] Note that m_0 is not forwarded to B if $B \in CORRUPT$, in that case B is fully incorporated in the network plus adversary.[17] The model has also been optimised to exploit the fact that the synchronous contract-signing protocol does not require a particular correct party to send the same message more than once to the same party (so there is no attempt to guarantee any delivery after the first).

Our $SYNC\_WITH\_SPY$ model can now be defined as follows:

[16]Note that this model permits so-called 'rushing adversary' attacks – in which the adversary can pass on a message, m', to any number of parties before the network has a chance to deliver m to the intended recipient. This is one of the reasons why a single *comms* channel – *comms.s.r.m* meaning s communicating m to r – will not suffice for synchronous communication models in general. Here both sender, s, and receiver, r, would have to synchronise on the *comms.s.r.m* event, and the spy would not learn of m before r.

[17]If one can defend an assertion that knowledge of m_0 does not add anything to the spy's ability to attack correct nodes, then, of course, we can omit the spy's sending of messages inferred from m_0 to correct parties.

$$
\begin{aligned}
SYNC\_WITH\_SPY \;=\; & [|tock|]\, A : P,\, B : P - \{A\},\, m_0 : Msg \;@ \\
& SYNC\_WITH\_SPY_{A,B,m_0}(false, 0, false)
\end{aligned}
$$

Figure 4.7: The complete CSP$_M$ model of the synchronous communications network and spy

However, we can optimise this model by first excluding the spy's initial knowledge from any the $Deductions(\{m_0\}, CORRUPT)$ message sets. We can then model the spy's sending of messages from its initial knowledge set in a separate process, rather than unnecessarily duplicating that behaviour across all the $SYNC\_WITH\_SPY_{A,B,m_0}(false, 0, false)$ processes.

In general it may not be possible to parallelize a $SYNC\_WITH\_SPY$ like the contract-signing version above. If not, then we must either share state between parallel processes (by synchronising on sets of events for example), or else write larger processes which retain 'predicate' parameters of set type.[18] FDR, however, does not efficiently compile sets, and sets with cardinalities of upwards of about five can cause compilation difficulties.[19] The use of renaming can help considerably when we have a large number of parallel symmetric processes all referencing sets. We can then specify the processes through renamings of one particular process, FDR has then only to compile this single process. This problem – of coding the spy's knowledge efficiently in general – is well known, and was the impetus for the development of the so-called 'lazy-spy' optimisations. In essence, the 'lazy spy' re-evaluates its knowledge 'on the fly', as opposed to evaluating all possibilities of its knowledge at compile time. The 'lazy-spy' is described fully in [53] and [54], for example.

Stop failures, in which fundamentally correct machines may intermittently stop all processing for a period of time, are easy to model in CSP. We could, for example, introduce a boolean-valued channel for each protocol. Each channel would indicate whether the process had 'stopped', or not. While a process is stopped, it engages in the *tock* events (but *without* counting them), and its *receive* and 'user input' events, both of which it ignores; if it is not stopped, then it performs its normal behaviour. The switch between 'stopped' and 'not stopped' states would generally be non-deterministic.

Secure channels - which we might choose to define as channels over which the spy cannot 'hear' messages - are also easy to model. In this case, we simply model the spy so that it never even engages in any events sent over the secure *send.A.B* - *receive.B.A* links. If we need to model both secure and insecure links between the same two principals, then we could, for example, introduce a boolean-valued field to the *send* and *receive* channels to indicate whether the link is assumed to be secure or not. In a similar vein, we could introduce an integer-valued field to account for multiple links between two principals.

[18]We have in mind here parameters of the form S, where $m \in S$ would equate to m having been overheard by the spy.

[19]Although the number of reachable states may actually be quite small, FDR would be predisposed - and, indeed, may even be obliged – to compile all possible state subsets whether they were reachable or not.

Authenticated channels are modelled so that a message is never delivered over that channel in some other principal's name. The CSP$_M$ of 4.6 actually defines an *unauthenticated* link between the A and B - as the spy may forward the message in its own name. However, this was irrelevant to the contract-signing protocol models, as they always engage in events over the *receive* channel anonomously in the *sender* field (i.e. their *receive* events are of the form $receive.A?\_?m : msg3$) - authentication is by way of payload analysis alone. Making the channel completely authenticated, however, would be a simple matter.

4.2.7 Optimizing the Message Space

We define a function $M : CORRECT \times P \rightarrow \mathbb{P}(Msg)$ so that $M(A, B)$, for example, is the set of messages that correct node A can potentially send to another node B (correct or not). The function M can be checked against the model of the protocol in question, for example, by checking that a contract-signing $SIGN(A, 0, \{\})$ process, with all but the *send* events of its alphabet hidden, is a traces refinement of $RUN(\, \{send.A.p.m \mid p \leftarrow P,\, m \leftarrow M(A, p)\}\,)$. If traces refinement holds, then it adds credence to the correctness of our CSP_M implementation - for quite complicated scripts – such as the contract-signing scripts - this is a cheap, but useful 'sanity check'.

For the synchronous contract-signing model, the function M can be used to fully optimize $SYNC\_WITH\_SPY$ as follows:

$$SYNC\_WITH\_SPY \;=\; [|tock|]\, A : P - CORRUPT,\, B : P - \{A\},\, m : M(A, B) \;@\; SYNC\_WITH\_SPY_{A,B,m_0}(false, 0, false)$$

where the $SYNC\_WITH\_SPY_{A,B,m_0}$ processes are defined as in Figure 4.6.

4.3 CSP Model of the Synchronous Contract-Signing Protocol

In this section we describe our analysis of the message-optimal synchronous contract-signing protocol (see Section 4.1.4). We begin the description of our CSP model by detailing the processes and datatypes. In Section 4.3.2 we argue that the results of our verification can be extrapolated to an arbitrary number of signatories, contracts and contract-signing sessions. In Section 4.3.4 we describe how the requirements on the synchronous contract-signing session are specified as CSP$_M$ refinement checks that can be run through FDR. In Section 4.3.3 we describe how to run the scripts on FDR. In Section 4.4 we report on modelling issues that arose during our analysis, and finally we present our conclusions and discuss avenues for further work.

4.3.1 The CSP Model of a Protocol Session

The CSP model of a synchronous contract-signing session is comprised of the communication network, the TTP (running 'resolve'), the Verifier, and the participating signatories.

We use our general model of the network, incorporating adversarial behaviour, as described in Section 4.2, to represent the communications network. The other components are protocol specific. Our model does not allow for a party entering into a contract-signing session either with itself, or with the TTP or Verifier.

4.3.1.1 Datatypes

Wherever possible the CSP_M scripts adopt the protocol original naming conventions from Section 4.1.4. Only the message names m_5' had to interchanged because the original m_5 equals m_2, while the original m_5' occurs much more often.

The datatype *machines* defines the machine identifiers:

```
datatype machines = A | F | B | TTP | V
```

where `A` and `B` are the identifiers of correctly functioning signatory machines, `F` is the identifier of an adversarial or malfunctioning signatory machine, `TTP` is the trusted third party's identifier, and `V` is the Verifier's identifier.

The signatory identifiers were ordered `A<F<B`. We imposed this ordering to demonstrate a simple method for automating agreement among the signatories as to who is to act as the 'sender' in a session. We stipulated that the 'least' of the two signatories is to act as 'sender'. It was necessary, then, that `A<F` and `F<B`. We could then ensure that we verify the two cases: (i) a correct signatory (i.e., `A`) acts as sender in a contract-signing session with the malicious agent `F`, (ii) the malicious agent acts as sender in a contract-signing session with a correct signatory (i.e., `B`). Note that this was a generalisation of the original protocol specification in which there were *separate* sender and receiver machines. The generalisation makes no difference to the formal verification other than to demonstrate the feasibility of automating the choice of sender and receiver (because A always acts as sender, and B as receiver).

The set *SIGs* contains the names of the signatories taking part in the contract-signing session, it is always a subset of $\{A,B,F\}$. The set of machines considered to be corrupted or malfunctioning are given by *CORRUPT*. This set is either $\{F\}$, or $\{F,TTP\}$ (the latter when we are verifying requirements which assume only limited trust in the TTP). For example:

```
nametype SIGs = {A,F}
CORRUPT = {F}
```

The faulty machine, F, may or may not be one of the those signatories.

The nametype P represents all the parties, less V, involved in a contract signing session between the signatories *SIGs*.

```
nametype P = union(SIGs,{TTP,F})
```

We excluded the Verifier's identity from P for expediency only. The Verifier machine is modeled so as to receive messages over the network, like the other machines, but it does not need to send messages. It only receives over the network communications channel, internally processes the received messages, and then outputs over a dedicated channel.

The nametype $C = \{\mathit{1..2}\ \}$ represents the contract texts, and the nametype $TID = \{\mathit{1..2}\ \}$ represents the transaction identifiers. The nametype $R = \{\mathit{0}\ \}$ is the set of starting-rounds.

These are additional simplifications for state-space reductions, see Section 4.3.2.

```
nametype C = {1..2}
nametype TID = {1..2}
nametype R = {0}
```

The maximum number of tocks over which a contract-signing session is played-out in our model is given by *MaxNoOfTocks*:

```
MaxNoOfTocks=7
```

The rational for the imposition of this bound may be found in the timing discussion preceeding the code for the *SIGN* model in 4.3.1.3

The signatory machines have user *input* and *output* channels. Input events are of the form *input.SIGs.sign.SIGs.C.TID* or *input.SIGs.show.TID*. For example, *input.A.sign.B.2.1* conveys a user request to signatory machine *A* to sign a contract with signatory machine *B*, with text *2* and transaction identifier *1*. *input.A.show.1* conveys a user request to *A* to show the status of the transaction with identifier *1* to the Verifier machine.

The datatype *msg3* represented the set of all 'signed' and 'unsigned' messages that can be passed between participating machines during the course of a contract-signing session. These are messages with at most three prefixed signatures. An example of a *msg3* message is:

```
S.TTP.S.B.S.A.U.A.B.0.2.1
```

If $p \in P$, then the string *S.p* prefixed to a message is read as machine *p* having 'signed' that message. The 'unsigned part' of this message is *U.A.B.0.2.1*, and is used to convey the details of a session. In this case those details are: initiated in round *0* with TID *1*, where *A* and *B* are to sign the contract with text *2*. The channels used to convey messages between the participating machines are *send* and *receive*, as described in Section 4.2.3.

A number of simple functions were defined for manipulating messages, or to reference particular message fields, for example:

1. $\mathit{Sender}: \mathit{Msg} \rightarrow P$ returns the content of a message's sender field.

2. $\mathit{ContractOf}: \mathit{Msg} \rightarrow C$ returns the content of a message's contract field.

3. $Pick(X)$ returns the unique member of any singleton set X.

4. $Sign : P \times Msg \rightarrow Msg$ 'signs' a message on behalf of a machine, p, say, by prefixing the message with the string $S.p$.

5. The ordering on the signatory identifiers is defined by the *Least* function $P \times P \rightarrow P$. $Least(p, q)$ is the least of p and q.

6. The function $Deductions : Msg \rightarrow \mathbb{P}\ (Msg)$ returns the set of messages that the adversary can deduce from a given message, less the adversary's initial knowledge.

7. $M : P\text{-}CORRUPT \times P \rightarrow \mathbb{P}(Msg)$ was defined so that if $p \in P\text{-}CORRUPT$ and $q \in P$, then $M(p, q)$ contains all the messages that p could conceivably send to q during the course of a contract-signing session.

4.3.1.2 The Communication Infrastructure and the Spy

We can use *SYNC_WITH_SPY* to model the synchronous communication infrastructure and the adversary's behaviour. That model is described in full in Section 4.2.6.

The CSP_M code for *SYNC_WITH_SPY* is as below, where the process `SYNC_A_B_M0(a,b,m0,know,n,received)` corresponds to $SYNC\_WITH\_SPY_{A,B,m_0}(...)$[20].

Note that the $SYNC\_A\_B\_M0$ processes only engage in *receive* events sent in the name of F. This is a legitimate optimisation as the contract signing protocol model, $SIGN$, only ever engages in events over the receive channel with anonymous sender field. This is because it cannot infer any reliable information from the sender field. The channels are 'unauthenticated', and authentication should be inferred from payload content only.

For efficiency, the spy's 'initial knowledge' (being all the unsigned messages, or messages signed only by the spy) was excluded from any $Deductions(m)$. The $SYNC2$ process accounts for the spy's ability to send messages from its 'initial knowledge' set at any time. This way we avoided unnecessary duplication of behaviour across all the $SYNC\_A\_B\_M0$ processes.

Note also that the *latency* in this case is *2* - i.e. a message is guaranteed to be delivered before 2 *tocks* have elapsed since it was sent[21].

```
CORRECT = diff(union(P,{V}),CORRUPT)
```

Calculate the correct machines.

[20]In $SYNC\_WITH\_SPY_{A,B,m_0}(...)$, the parameters a, b and m were supposed fixed. This was to leave open the opportunity of more efficient compilation. The idea being to compile the single $SYNC\_WITH\_SPY_{A,B,m_0}(...)$ for a fixed $a = A$, $b = B$ and m_0, and then rename. However, this compilation technique is prone to error unless the *Deductions* function is very simple. This is not the case here, and so the a, b and m_0 were brought into the parameter list for `SYNC_A_B_M0`.

[21]This helped in the debugging of the model using the Probe CSP animator. A latency of 1, however would have sufficed.

```
SYNC_A_B_M0(a,b,m0,know,n,received) = (

   send.a.b.m0 ->
   if member(b,CORRECT) then
      SYNC_A_B_M0(a,b,m0,true,n,received)
   else
      SYNC_A_B_M0(a,b,m0,true,0,true) )

[] know==true and received==false  & (
   receive.b.F.m0 -> SYNC_A_B_M0(a,b,m0,know,0,true) )

[] know==false or received==true & (
   tock -> SYNC_A_B_M0(a,b,m0,know,0,received) )

[] know==true and (received==false and n<1) & (
   tock -> SYNC_A_B_M0(a,b,m0,true,n+1,received) )

[] know==true & (
   receive?r:CORRECT!F?m:inter(Deductions(m0),msg3) ->
   if r==b and m==m0 then
      SYNC_A_B_M0(a,b,m0,know,n,true)
   else
      SYNC_A_B_M0(a,b,m0,know,n,received) )


InitialKnowledge = {m|m<-msg3,Signatories(m)=={F} or Signatories(m)=={}}

SYNC2 =

( receive?_:CORRECT!F?_:InitialKnowledge -> SYNC2 )

[] ( tock -> SYNC2 )



SYNC_WITH_SPY =
( [|{tock}|] a:diff(CORRECT,{V}), b:union(OtherThan(a),{V}),
             m:inter(msg3,M(a,b)) @
   SYNC_A_B_M0(a,b,m,false,0,false) )
[|{tock}|]
SYNC2
```

4.3.1.3 Protocol Specific Processes

There are three protocol-specific processes included in our model: the process modelling runs of the contract-signing protocol by correct signatories (participants), `SIGN(this,r,cache)`; the 'resolve' protocol as run by a correct trusted third party, `RESOLVE`; and the verification process, `VERIFIER`.

The `SYNC(this,0,{})` process is a model of `this` signatory's correct execution of the message-optimal synchronous 'sign' protocol.

`SYNC` has parameters `this`, `r`, and `cache`. `this` is the identifier of the machine running the protocol. `r` keeps count of the number of `tock`s that have elapsed since the user request to sign the contract was received. `cache` is a singleton set recording the last message sent (if any)[22].

The protocol as modelled here was generalized so that a correct signatory can react to a possibly early receipt of an expected message by sending earlier. In the original protocol, the signatory *always waits* to the next round before it responds by sending a message back to a co-signatory. From the information stored in `cache`, a correct signatory is able to ascertain what round (of the original protocol specification) the session is currently in (although the round number in this generalised version will not neccessarily correspond to the `tock` count).

The process will wait until it receives a user request to sign a contract with another signatory. Prior to that it is prepared to engage in `tock` (although the clock count, `r`, will not yet be incremented), but any messages received over the network are ignored.

User requests are conveyed to `SIGN(this,0,{})` through events of the form:

`input.this.sign?p:diff(SIGs,{this})?c:C?t:TID`

The first such event that the `SIGN(this,0,{})` process synchronizes on will dictate the transaction identifier, `t`, of the session that `SIGN(this,0,{})` is to play out.

Once it has received a user request to sign a contract, a correct party ascertains who is to be the sender, and who is to be the receiver in the session. The least of the signatories, where 'least' is defined by the `Least` function, is to act as sender.

A correct sender will then send the first of two messages, `m1`, to the sender, recall Figure 4.1. A correct receiver, on the other hand, will record the fact that a session has begun and wait for `m1` In the optimistic case, two more messages will then be sent between the two signatories as the session is played out. On receipt of `m1` a correct receiver will send a message `m2` back to the sender. On receipt of `m2`, the session is brought to conclusion by a correct sender sending a message `m3` back to the receiver.

On receipt of `m2` (which may have been forwarded by the trusted third party as part of the resolve protocol), a correct sender is able to declare the contract as 'signed'. It does this by engaging in the event:

[22]Used simply, as they are here, sets are not a problem for FDR. They are badly compiled when, for example, the union of sets appears in recursion, such as $Spy(myknowledge) = \cdots \rightarrow Spy(union(myknowledge, \{m\}))$.

`output.this.signed.tid_of(m2)`

A correct receiver is only able to declare the contract 'signed' once it has received `m3`, or an affidavit, `m5` sent by the trusted third party as part of the resolve protocol.

Things may go wrong when the malicious agent is one of the signatories. If the sender is correct, then it may not receive the expected `m2` or `m5_prime` (i.e. $m5'$) message by the end of round five. In that case it will reject the session by outputting `output!this!rejected!t`. If the receiver is correct, then it may not receive `m1` by round two, in which case it rejects the session, or it may not receive `m3` by round four, in which case it starts the resolve protocol, which is described below.

Provided that it is in the 'signed state', then the `SIGN(this,r,cache)` process is prepared to engage in a user request to 'show' the status of a contract-signing session to the Verifier machine. Such a request will be conveyed to `SIGN(this,r,cache)` via the `input!this!show?t:TID` events. The `SHOW(this,r,cache,t)` process is described in detail below.

The timing of the sending and receiving of the messages `m1`, `m2`, `m3`, `m4`, `m5` and `m5_prime` are as follows. Recall that a latency of 2 guarantees that a message will be delivered before at most 2 *tocks* have elapsed since it was sent.

`r` is not incremented prior to the first user request to 'sign'. All tocks that occur subsequent to this first user request to sign, result in `r` being incremented until `r`=7 at the maximum. The rational for the imposition of this particular bound on `r` is given below.

`m1` is sent when `r`=0, and is guaranteed to be delivered at any time while `r`<2.

A correct receiver will reject the session only when it can be certain that `m1` was never sent, i.e. immediately after `r` becomes 2.

If `m1` was received by a correct receiver, then `m2` will be sent any time while `r`<2, and will be delivered any time while`r`<3.

`m3` is sent any time while `r`<3 and will be delivered any time while `r`<4.

A correct receiver must wait until `r`=4 until it can be certain that `m3` was not sent. If `m3` has not been received by the time `r` becomes 4, then it immediately sends `m4` to the `TTP` in order to start the resolve process.

`m4` will be delivered to the `TTP` while `r`<6. A correct `TTP` will then check that the received message is in the correct format, and that it was not sent later than expected (to guard against a correct sender having already rejected the session). If so, the `TTP` sends `m5_prime` and `m5` to the sender and receiver respectively.

`m5_prime` and `m5` will be received sometime while `r`<7.

Only when `r` becomes 7 can a correct sender assume that it will never receive `m2` or `m5_prime`, if it has not already done so, in which case it will immediately reject the session.

Similarly, a correct receiver, if it did need to resort to the `TTP`, will receive `m5` while `r`<7.

Any message that a correct party receives over the communications network when `r`≥7 is therefore discarded. The only correct activity which occurs when `r`≥7 is the possible output of a 'rejected' or a 'signed' by correct senders or receivers, and requests to run the ‘show’ protocol on a user request. We can therefore stop counting tocks at `r`=7.

The CSP$_M$ code for `SIGN(this,r,cache)` is as follows:

```
is_m4(b,m) =
if not member(m,msg3) then
   false
else if member(m,{|S.b.S.b.S.Sender(m).U.Sender(m).b|}) then
   true
else
   false
```

`is_m4(x,m)` returns true if m is of the form of a ‘m4’ message that a correct receiver `b` might send to the `TTP` in order to begin the resolve process. Else it returns false.

```
SIGN(this,r,cache) =

   if r==7 and (not empty (inter(cache,diff(msg1,msg0))))
                and Sender(Pick(cache))==this then (

      -- A contract-signing session was played-out with
      -- ‘this’ machine as sender, but neither ‘m2’ nor
      -- ‘m5’ have been received by the end of round five.
      -- Reject the session.

      let
         t = tid_of(Pick(cache))
       within (

          output!this!rejected!t ->
          SIGN(this,r,{t.U} ) ) )

   else if r==4 and (not empty(inter(cache,diff(msg2,msg1))))
                        and Receiver(Pick(cache))==this then (

      -- If by round four, ‘this’ receiver machine was
      -- expecting, but has not received message ‘m3’, then
      -- start the ‘resolve’ protocol.

      let
          m4 = Sign(this,Pick(cache))
      within (

         -- Send ‘m4’ to the TTP.
         send!this!TTP!m4 -> SIGN(this,r,{m4}) ) )
```

```
else if r==2 and (not empty(inter(cache,msg0))) and
                 Receiver(Pick(cache))==this then (

    -- ‘m1’ was never received by ‘this’ receiver
    -- machine  - so reject the session.

    let
       t = tid_of(Pick(cache))
     within (

        output!this!rejected!t ->
        SIGN(this,r,{t.U} ) ) )

 else (

empty(cache) & (

    -- Wait for an input requesting that ‘this’ sign
    -- a contract.

    input.this.sign?p:diff(SIGs,{this})?c:C?t:TID ->

    -- A contract, ‘c’, is to be signed with transaction
    -- identifier ‘t’. The co-signatory is ‘p’.

    if Least(this,p)==this then (

       -- ‘this’ is to play the part of sender in the contract
       --  signing session.

       let
             m1 = Sign(this,U.this.p.0.c.t)
       within (

          -- Send ‘m1’ to the co-signatory, p, and cache ‘m1’.

          send!this!p!m1 -> SIGN(this,r,{m1}) ) )

    else (

       -- The co-signatory, ‘p’, will be the sender in the
       -- contract signing. Await the receipt of  ‘m1’.

       SIGN(this,r,{U.p.this.r.c.t}) ) )
```

```
   -- If in ‘signed state’ (cache contains a message with 3 signatures),
   -- then run ‘show’ protocol when requested by the user
   [] card(inter(cache,diff(msg3,msg2)))==1 & (
      input!this!show?t:TID -> SHOW(this,r,cache,t) )


[] card(diff(cache,msg2))==1 and (not is_m4(this,Pick(cache))) & ( (

      -- cache contains either ‘m3’, ‘m5’ or ‘U.t’ ( ‘U.t’ is
      -- used to indicate that the session with TID ‘t’ was
      -- rejected). So, a correct machine’s part in the protocol
      -- run is over. It will send no more messages, and all
      -- messages received hereafter are ignored.

         receive!this?_:OtherThan(this)?_:msg3 ->
         SIGN(this,r,cache) ) )

[] r<MaxNoOfTocks and not empty(cache) & (

      -- A session is underway with time to spare.
      -- If a ‘tock’ does occur, then increment
      -- the timer counter r.

      tock ->  SIGN(this,r+1,cache) )


[] r==MaxNoOfTocks or empty(cache) &  (

      -- A session has finished, or we are still
      -- waiting for the user request to sign.
      -- Either way, if a ‘tock’ does occur, don’t
      -- increment the timer counter r.

      tock ->  SIGN(this,r,cache) )


[] r<=MaxNoOfTocks and inter(cache,msg2)!={} or
  (cache!={} and is_m4(this,Pick(cache))) & (

      -- A contract signing session is underway
      -- with time to spare.Must process any
      -- message received.

      let
   u = UnsignedPart(Pick(cache))
   p = CoSig(this,u)
   t = tid_of(u)
within (

   receive!this?_:OtherThan(this)?m:msg3 ->

   if tid_of(m)!=t or member(m,msg0) or
      diff(Signatories(m),{this,p,TTP})!={} then (

       -- Discard  the message - it does not pertain to
       -- this session or is part of a replay attack

      SIGN(this,r,cache) )

   else if u==Pick(cache) and member(m,diff(msg1,msg0))
   and tid_of(m)==tid_of(u) and ContractOf(m)!=ContractOf(u) then (

      -- A possibe ‘m1’ was received, but the contracts do not
      -- match - so reject the session.

      output!this!rejected!t -> SIGN(this,r,{t.U}) )

   else if (S.p.Pick(cache))==(m) or (S.TTP.S.this.S.p.u)==(m) then (

      -- ‘m’ is the expected message.

      let
         n = no_of_signatures(m)
      within (

         if n==1 then (

            -- ‘m’ is ‘m1’.

            let
               m2 = Sign(this,m)
               -- Sign ‘m1’.
            within (

               -- Send ‘m2’ to the co-signatory,
               -- cache ‘m2’ and go to round r+1

               send!this!p!m2 -> SIGN(this,r,{m2}) ) )
```

```
        else if n==2 then (

           -- ‘m’ is ‘m2’.

           let
              m3 = Sign(this,m)
              -- Sign ‘m2’
           within (

              -- Output ‘signed’, send ‘m3’ to
              -- the co-signatory, cache ‘m3’ and
              -- go to round r+1.

              output!this!signed!tid_of(m) ->
              send!this!p!m3 ->
              SIGN(this,r,{m3}) ) )

        else (

           -- ‘m’ is ‘m3’ or ‘m5’ output ‘signed’ and
           -- cache ‘m’.

           output!this!signed!tid_of(m) ->
           SIGN(this,r,{m}) ) ) )

  else (

     -- Discard  the message - it is a
     -- possible replay attack.

     SIGN(this,r,cache) ) ) ) )
```

The `SIGN(this,r,cache)` process recurses to `SHOW(this,r,cache,t)` when it is in the ‘signed’ state, and it receives a user request to ‘show’ the status of a contract-signing session to the Verifier machine. The user show requests are conveyed to `SIGN(this,r,cache)` by synchronizing on `input!this!show?t:TID` events, after which the `SHOW(this,r,cache,t)` protocol is called:

```
[] (  input!this!show?t:TID -> SHOW(this,r,cache,t) )
```

The `SHOW(this,r,cache,t)` protocol will then look to see whether `cache` contains an `m3` or `m5` message pertaining to the session `t`. If so, then that message is forwarded to the VERIFIER machine over the network, and then SHOW(this,r,cache,t) recurses back to SIGN(this,r,cache). If there is no m3 or m5 message in cache pertaining to t, then SHOW(this,r,cache,t) immediately recurses back to SIGN(this,r,cache).

The CSP$_M$ code for SHOW(this,r,cache,t) is as follows:

```
-- THE SHOW PROTOCOL

-- The ‘show’ protocol as called by ‘this’
-- correctly behaving machine. Look up m3
-- or m5 and send to Verifier.

SHOW(this,r,cache,t) = (

   let
      X = { x  | x<-inter(cache,diff(msg3,msg2)), tid_of(x)==t }
   within (
      if not empty(X) then (
         send.this.V.Pick(X) ->
         SIGN(this,r,cache) )
      else (
         SIGN(this,r,cache) ) ) )
SHOW(this,r,cache,t) = (
```

The VERIFIER machine receives messages from the SHOW(this,r,cache,t) protocol over the network via receive.V._?m:msg3 events. When it receives a message m, it must check whether or not m is the product of a successful contract-signing session. The test for this is that m be an m3 or an m5 message. An m3 message will have the form expected of the last message that would be sent by a correct sender to the recipient during the course of a contract-signing session in which there was no need to resort to the resolve protocol. An m5 message will have the form expected of an affidavit sent to a receiver by the TTP machine as part of the resolve protocol. If m is of the form m3 or m5, then the VERIFIER will output outputV.Signed.a.b.c.t, indicating the success of the contract-signing session (where a, b, c and t are, respectively, the two signatory identifiers, the contract, and the transaction identifier to which the message pertains). If m is not an m3 or a m5, then VERIFIER will output outputV.Rejected.t, where t is the transaction identifier to which the message pertains.[23]

The CSP$_M$ code for VERIFIER is as follows:

```
-- THE VERIFIER

-- A model of the Verifier machine.
```

[23]We had to introduce Signed and Rejected for the Verifier's output, as opposed to reusing signed and rejected, for technical reasons regarding CSP$_M$'s datatype syntax.

```
VERIFIER = (

   receive.V?_?m:msg3 ->
   let
      t = tid_of(m)
   within (
      if (no_of_signatures(m)<3) then (
         outputV!Rejected!t ->
         VERIFIER )
      else (
         let
            u = UnsignedPart(m)
            a = Sender(u)
            b = Receiver(u)
            c = ContractOf(u)
         within (
            if (m)==(S.a.S.b.S.a.u) or
               (m)==(S.TTP.S.b.S.a.u) then (
                -- m==m3 or m==m5
               outputV!Signed!a!b!c!t->
               VERIFIER )
            else (
               outputV!Rejected!t ->
               VERIFIER ) ) ) ) )

   [] ( tock -> VERIFIER )
```

The `RESOLVE` protocol is run by the `TTP` machine in the event that a correct receiver does not receive the second expected message by round four. In that case, a correct receiver will send to the `TTP` a message, `m4`. This message will be of form `S.b.S.b.S.a.U.a.b.r.c.t` where, respectively, `b` and `a` are the (receiver and sender) signatory identifiers, `r` is the start-time (always 0), `c` is the contract, and `t` is the transaction identifier. If an `m4` is received by the `TTP`, then the trusted third party checks that it was not sent late by a possibly malicious receiver. The check for this, as defended above, in the timing analysis, is that `r`-0 be less than 6. If the received message is of the correct format, and the timing is right, then `TTP` will send an affidavit, `m5` to the receiver, and `m5_prime` to the sender, where `m5=S.TTP.b.U.a.b.r.c.t` and `m5_prime=S.b.S.a.U.a.b.r.c.t` are derived from the `m4` message. The `TTP` will discard any messages that it receives which are not of the form `m4`, or were sent late.

The CSP$_M$ code for `RESOLVE` is as follows:

```
-- THE RESOLVE PROTOCOL
-- The  ‘resolve’ protocol as executed by
-- a correctly functioning TTP.

RESOLVE(r) = (

   receive!TTP?_:OtherThan(TTP)?m:msg3 ->
   let
       u = UnsignedPart(m)
       a = Sender(u)
       b = Receiver(u)
       r0 = r0Of(u)
       m2 = S.b.S.a.u
       m4 = S.b.S.b.S.a.u
   within (
      if (m)==(m4) and (r-r0)<6 then (
         -- ‘m’ is of the correct format,i.e. an ‘m4’, and was
         -- not sent late.
         let
            m5 = Sign(TTP,m2)
         within (
            -- Send m5’==m2 to ‘A’
            send!TTP!a!m2 ->
            -- Out affidavit, m5, to ‘B’
            send!TTP!b!m5 ->
            RESOLVE(r) ) )
       else
          RESOLVE(r) ) )

[] r==MaxNoOfTocks & ( tock -> RESOLVE(r) )

[] r<MaxNoOfTocks & (tock -> RESOLVE(r+1) )
```

4.3.2 Extrapolation and Abstraction Arguments

We assume that the underlying cryptographic signature scheme is unforgeable. In our abstracted cryptographic scheme, the prefixing of a message m by the string $S.p$ corresponds to signatory p having signed m. The cryptographic signature scheme assumes that one may verify whether or not any such signature appended to m genuinely pertains to m. This translates as it being computationally infeasible for the spy to infer solely from message m, any messages other than those messages m' which are such that after deleting any number (possibly zero) of the leading signatures of m' pertaining to corrupted parties, the resulting string is a suffix of m'. Any other manipulation of the message would be detectable by a correct principal, who could then discard it. As a result, the messages that a spy may infer from $\{m1,m2\}$ is no more than what he can infer from $m1$ plus what he can infer from $m2$.

The protocol we are analysing is designed to operate correctly for any number of principles, contracts, and sessions. We cannot check an arbitrary number of any parameters directly using FDR. However, for the purpose of our analysis we argue that it is sufficient to verify, separately, a number of single runs of a contract-signing session, between the combinations of two signatories chosen from $\{A,B,F\}$.

A particular run of the contract-signing protocol, as played out by a correct machine, begins when the machine first receives a user-request to sign a contract. Such a request is supplied with what will be the contract-signing session's unique transaction identifier (*TID*). Any subsequent message that is either not correctly formatted, or which pertains to a different *TID* can be safely ignored. This is because either the message is nonsensical, or it is irrelevant to the session in question. Therefore, sessions are logically distinct, so it is sufficient for us to consider one session, where messages are either part of that session, or not.

We use a type of $\{1..2\}$ for the TIDs, a type of $\{0\}$ for the initial-round numbers, and a type of $\{1..2\}$ for the contract texts, C.

We limited the *TID* datatype to a type of cardinality two as this is the threshold on the cardinality of the datatype required to infer data independance in the formal Roscoe/Lazic Data Independance theory[24]. The same is true for the contract datatype. The initial round numbers could, in theory, be modelled data independantly, but it would be highly complex to do so[25].

We must verify six individual 'base-case' runs of the contract-signing protocol in order to ensure complete coverage of behaviors. The protocol behaviour is asymmetric in sender and receiver signatories. So, our base-cases are for contract-signings between the two correct signatories, and between one correct signatory and one incorrect signatory in the cases when the correct signatory acts as sender, and when the incorrect signatory acts as sender. Part of the requirements stipulate that we must consider each of these contract-signing sessions in the presence of either a reliable, or an unreliable TTP.

4.3.3 Running the Scripts

The compilable CSP$_M$ scripts for the synchronous-signing protocol are to be posted in the public section of the MAFTIA homepage at [31]. It is intended that all CSP$_M$ models developed under WP6 will eventually be posted on this site for download.

There are nine .csp scripts in total.

The *types.csp* script contains the datatype, nametype and function definitions.

The *sync.csp* script contains the $SYNC$ generic model of the communication network plus adversary.

[24]The transaction identifier's are only ever compared for equality.

[25]Amongst other things, it would neccessitate ordering {0..MaxNoOfTocks} non-deterministically as part of the modeling of a run of the protocol.

The *protocol.csp* script contains the *SIGN*, *SHOW*, *RESOLVE* and *VERIFIER* models.

In addition, there are six 'base-case' *test* .csp scripts.

Each of the *test* scripts defines a set of signatories, *SIGs*, plus the set of 'corrupted' parties, *CORRUPT*. These cases corrspond to the different trust models that need to be covered.

It is the *test* scripts that must be loaded into FDR. Each of the *test* scripts includes the *types.csp*, *sync.csp* and *protocol.csp* scripts, and at this point the types P and *Msg* are instantiated from *SIGs* and *CORRUPT* as defined by the presently loaded *test* script.

The *test* scripts 1, 2 and 3 all assume a reliable TTP (i.e., $CORRUPT = \{F\}$). The *test* scripts 4, 5 and 6 all assume an unreliable TTP (i.e., $CORRUPT = \{F, TTP\}$).

test1.csp and *test4.csp* are designed to verify runs of the protocol between a correct sender, i.e., A, and a malicious receiver, i.e., F.

test2.csp and *test5.csp* verify runs of the protocol between a correct receiver, i.e., B, and a malicious sender, i.e., F.

test3.csp and *test6.csp* verify runs of the protocol between the two correct players, i.e., A and B, in the presence of the spy, F.

4.3.4 Modelling the Requirements

There are seven security and liveness requirements that are expected to be fulfilled by the synchronous contract-signing protocol. The requirements are referred to according to their names in [55]:

- *Requirement 4.2a (Correct Execution)*
- *Requirement 4.2b (Unforgeability of Contracts)*
- *Requirement 4.2c (Verifiability of Valid Contracts)*
- *Requirement 4.2d (No Surprises with Invalid Contracts)*
- *Requirement 4.2e (Termination on Synchronous Network)*
- *Requirement 4.3b (Limited Trust in T)*
- *Requirement 4.3c (Optimistic on Agreement on Synchronous Network)*

These requirements were specified in CSP$_M$ through a number of CSP$_M$ assertion tests which we describe below. Note that some of these CSP$_M$ tests vary according to the identifiers of the correct signatories under consideration by a particular test script. In the more complicated of these cases, we have stated the assertion tests for one of the correct signatories only for the sake of brevity .

In these tests, `CONTRACT` is the whole contract-signing model, and `alpha_CONTRACT` is its alphabet.

Some of the tests refer to processes C_X(S). A process C_X(S) is 'chaotic' behaviour in the alphabet of alpha_CONTRACT less S, where S will be some subset of alpha_CONTRACT[26]. In the traces model, one may read C_X(S) as doing any event at any time, other than those in S.

We also make much use of the CSP_M 'interrupt' binary process operator, /\. If P and Q are processes, then $P/\backslash Q$ means begin behaving like P for an arbitrary period of time (perhaps forever), but, possibly, stop behaving like P at any time, then begin behaving like Q.

Requirement 4.2a This requirement considers two correct signatories, A and B, executing the sign protocol after receiving user commands input.A.sign.B.c1.t and input.B.sign.A.c2.t, respectively. If the user inputs were in the same round, and the TID, t, was fresh, then both signatories will output rejected.t iff c1=c2, else both will output signed.A.B.c.t.

This requirement can be expressed in CSP_M by obliging the CONTRACT process to engage in the two input._.sign events in the same round. Then we assert that if the contracts are the same, then no output._.rejected.t events occur, and, if the contracts are not equal, then no output._.signed.t events occur. The fact that the contrary outputs must then occur, will follow from the verification of Requirement 4.2e.

```
assert C_X({|output.A.rejected.1, output.B.rejected.1|}) [T=
CONTRACT [|{|input,tock|}|] (
   RUN({tock}) /\ (
   input?a:SIGs!sign?b:diff(SIGs,{a})!1!1 ->
   input!a!sign!b!1!1 ->
   RUN({tock}) ) )

assert C_X({|output.A.signed.1, output.B.signed.1|}) [T=
CONTRACT [|{|input,tock|}|] (
   RUN({tock}) /\ (
   input?a:SIGs!sign?b:diff(SIGs,{a})!1!1 ->
   input!a!sign!b!2!1 ->
   RUN({tock}) ) )
```

Note that as the protocol is symmetric in the transaction identifier type, and in the contract types, so we could fix the values of the transaction identifiers and contracts in the tests above. If those assertions hold, then we can infer that the assertions would hold for the other possible combinations.

[26]It was defined as CHAOS(diff(alpha_CONTRACT,S)), where CHAOS is a predefined FDR process.

Requirement 4.2b This requirement states that if a correct signatory `A` has not far not received a user sign request, `input.A.sign.B.c.t`, say, then no correct `VERIFIER` will output `signed.A.B.c.t`.

For the CSP$_M$ refinement test, the `input.A.sign.B.c.t` event of `CONTRACT` was prevented from occurring. This was achieved by putting `CONTRACT` in shared parallel with `STOP`, obliging the two processes to synchronise on the `input.A.sign.B.c.t` event only. As `STOP` engages in no events, so `CONTRACT` would be free to do anything other than engage in `input.A.sign.B.c.t`. It was then asserted that the restrained behavior of `CONTRACT` was traces-refined by `C_X({|outputV.Signed.A.B.c.t|})`, i.e., any event but the `outputV.Signed.A.B.c.t` event could occur:

```
assert C_X({|outputV.Signed.A.B.1.1|})
[T= CONTRACT[|{|input.A.sign.B.1.1|}|] STOP

assert C_X({|outputV.Signed.A.B.1.1|})
[T= CONTRACT[|{|input.B.sign.A.1.1|}|] STOP
```

Requirement 4.2c This requirement states that if a correct signatory, `A` say, output `output.A.signed.t` on `input.A.sign.b.c.t`, and later executes show on input `t`, then a correct Verifier will output `outputV.Signed.A.b.c.t`.

This requirement is verified as follows. We hide all events of `CONTRACT` except `H(A)={|outputV, input.A.show.1, output.A.signed.1,tock|}`. We then assert that all visible behaviours of the resultant process are behaviours of `Req42c(A)`. The behaviour of `Req42c(A)` is as follows. It begins with chaotic behaviour in the events of `H(A)-{output.A.signed.1}`. When, if ever, an `output.A.signed.1` does occur, then it engages in chaotic behaviour in the `{|tock,outputV|}` events only, until, if ever, an `input.A.show.1` occurs. Its behaviour after that it is chaotic in `{|tock, outputV,input.A.show.1}` with the proviso that at most one tock elapses before an `outputV.Signed.A_._.1` event is engaged in. That is, if an `output.A.signed.1` event occurs, followed sometime later by a `input.A.show.1`, then the Verifier will output signed before two `tocks` elapse. Similarly for B.

```
Req42c(a) =

RUN({|tock,outputV,input.a.show.1|}) /\ (
output.a.signed.1 -> (
RUN({|tock,outputV|}) /\ (
input.a.show.1 ->
Req42c1(a,false,0) ) ) )
```

```
Req42c1(a,fin,n) = (
fin==true & (tock -> Req33c1(a,true,n)))
[] ( outputV.Signed.a?_?_.1 -> Req42c1(a,true,n) )
[] ( outputV.Signed?_.a?_.1 -> Req42c1(a,true,n) )
[] ( outputV.Signed?_?_?_?_:diff(TID,{1}) -> Req33c1(a,fin,n) )
[] n==0 & ( tock -> Req42c1(a,fin,1) )
[] ( outputV.Rejected?_ -> Req42c1(a,fin,n) )
[] ( input.a.show.1 -> Req42c1(a,fin,n) )

H(a) = {|outputV,tock,input.a.show.1,output.a.signed.1|}

assert Req42c(A) [T= CONTRACT \ diff(alpha_CONTRACT,H(A))

assert Req42c(B) [T= CONTRACT \ diff(alpha_CONTRACT,H(B))
```

Requirement 4.2d This requirement states that if a correct signatory `A`, say, at some point outputs `output.A.rejected.t`, then a correct `VERIFIER` will never have, and never will output `signed.a.b.c.t` where `a` or `b` is `A`.

We verified this requirement by asserting that there are no behaviours of `CONTRACT` in which an `outputV.Signed.a.b.c.1` occurs and also an `output.a.rejected.1` where `a=A` or `b=A`

```
Req42d =
C_X({outputV.Signed.a.b.c.1|a<-SIGs,b<-diff(SIGs,{a}),a==A or b==A,c<-C})
[]
C_X({output.A.rejected.1})

assert Req42d [T= CONTRACT
```

Note that for this requirement always to be fulfilled, the TTP may sometimes need to check whether or not an `m4` message sent is not sent *late* with respect to the initial start-time of a session (i.e. 0). Thus, that the above formulation allows the `TTP` to start counting `tocks` (up to 7, at which it stops) *before* a session has began is not relevant (although all strictly correct behaviours - in which the `TTP`'s 'clock' is synchronized with the signatories are played-out).

Requirement 4.2e This requirement states that a correct signatory `a`, on `input.a.sign.b.c.t` will output either `output.a.rejected.t` or `output.a.signed.t` after a fixed period of time. The `Req42e(false,0,false)` process begins by engaging in `tock` events only. When, if ever, an `input.a.sign.b.c.t` event occurs, then there are at most seven more tocks before an `output.a.signed.t` or a `output.a.rejected.t` event is engaged in. After which anything can happen. The assertion tests that `CONTRACT` with all except the `tock` and relevant `input.s` and `output.s` events hidden satisfies this behaviour. The deadlock-freedom test is necessary to ensure that `CONTRACT` never has the option of just doing nothing, and, therefore, trivially satisfying the assertion test (as, indeed, `STOP` would). That is, assuming time passes, as indicated by the visible `tock` events, the desired `output.s.signed` or `output.s.rejected` event will happen. See Section 4.4.4 for a full discussion on eventual termination.

```
upper = 7
-- The upper bound on the number of tocks, after which
-- a ‘send’ or ’rejected’ must be output.

Req42e(start,n,finished) =

   if start==false then (

      ( tock -> Req42e(false,0,false) )

      [] ( output.A?_ -> Req42e(false,0,false) )

      [] ( input.A.sign.F.1.1 -> Req42e(true,0,false) ) )

   else (

      n<upper & ( tock -> Req42e(true,n+1,finished) )

      [] ( input.A.sign.F.1.1 -> Req42e(true,n,finished) )

      [] finished==true and n==upper & ( tock -> Req33e(true,upper,true) )

      [] ( output.A.signed.1 -> Req42e(true,n,true) )

      [] ( output.A.rejected.1 -> Req42e(true,n,true) ) )


assert Req42e(false,0,false)
[T=
CONTRACT\diff(alpha_CONTRACT,{|
   input.A.sign.F.1.1,
   output.A.signed.1,
   output.A.rejected.1,
   tock|})
```

There are similar tests for other combinations of signatories, which we have omitted for the sake of brevity.

Requirement 4.3b This requirement states that 4.2b and 4.2c are fulfilled even if the TTP is incorrect (corrupt). It is verified in CSP$_M$ by re-running the assertions of 4.2b and 4.2c above in test scripts 4, 5 and 6, where `CORRUPT={TTP,F}`. Further details can be found in Section 4.3.3.

Requirement 4.3c This requirement states that if the two correct signatories, `A` and `B`, in the same round receive user commands `input.A.sign.B.c.t` and `input.B.sign.A.c.t`, respectively, with fresh TID, `t`, then the TTP does not send or receive messages to/from the correct signatories during the run of the sign protocol.

The requirement is specified in CSP$_M$ by obliging `CONTRACT` to engage in the two `input` events in the same round. The resulting behaviour is asserted to be traces-refined by `C_X({|send.TTP.a,send.a.TTP|a<-SIGs|})`, which, if true, confirms that the `TTP` never sends to, or is sent messages from `A` or `B`.

```
assert C_X({|send.TTP.a,send.a.TTP|a<-SIGs|})
[T=
CONTRACT [|{|input,tock|}|] (
   RUN({tock}) /\ (
   input?a:SIGs!sign?b:diff(SIGs,{a})!1!1 ->
   input!a!sign!b!1!1-> RUN({tock}) ) )
```

4.4 Results and Special Issues

4.4.1 Protocol Inaccuracies

One ambiguity of the original protocol formulation was found by FDR. The "notations and assumptions" section of the protocol contains the simple sentence "In the synchronous case, messages that do not arrive in their designated round are ignored." This is particularly important for the message m_4 sent to the TTP, because the signatory A only waits for a potential response m_5 until the next round. (This is the main reason why synchronous protocols can use fewer rounds than asynchronous ones, and the fact that TTP verifies the correct arrival time is used in the original proof of the requirement "no surprises".) Nevertheless, this test was forgotten in the original CSP formulation. This can be seen as yet another confirmation that protocol descriptions that are not totally explicit with respect to tests are apt to be misread, even in cases where the general rule to be applied has been stated somewhere (cf. [3]).

Another ambiguity arose in the formalization of the requirement "no surprises" because of the phrase "If a correct signatory, say A, outputs (rejected, *tid*), . . . , then no correct verifier will output . . . ". While the intention was to express that that the verifier should not accept the contract at all if the signatory gets the output that it has not been signed, it was first formalized temporally because of the "will", i.e., the verifier would have been allowed to accept the wrong contract during the signing protocol. (As this resulted in a weaker requirement, it could only be found by proofreading, not by FDR.)

4.4.2 Heterogeneous Network Topologies and Hybrid Timing Models

The $SYNC$ model described here is, in theory at least, quite a flexible model. For example, if we wished to change the latency of the A-B link on our network, then we would need only change *latency* in the $(received == false \, and \, n < latency)$ guards of all the $SYNC_{A,B,m}$ and $SYNC_{B,A,m}$ processes[27]. On paper, this is simple, but in practice one should not underestimate the effect that such an apparently minor modification can have on the state-space of the model. For the purposes of a future verification of large parts of the middleware, such simple changes are unlikely to suffice. If this is so, it will be necessary to investigate ways of modeling hybrid timing, or partial synchrony much more efficiently.

By parametrizing over sets other than $P.P.Msg$ in the protocol of Figure 4.5 in Section 4.2.4, $SYNC$ admits more complex topologies. For example, by parametrizing over $\{p.q.m \mid p \leftarrow P,\ q \leftarrow P, p \neq A \ or \ q \neq B,\ m \leftarrow Msg\}$ (rather than $P.P.Msg$), we get a $SYNC$ communication network with no A-B link. [28] ,[29]

4.4.3 Compilation Issues and State-Space Explosion

FDR's task of verifying a CSP model consists of two consecutive phases:

1. *the compilation phase*, in which the CSP model is given an internal representation as a state machine.
2. *the state exploration phase*, in which all the reachable states of the state machine are checked.

An inherent problem of model-checking tools, such as FDR, is that the verification task can become intractable. This intractability can occur in either of the two phases mentioned above.

The compilation engine in FDR is much slower than the state exploration engine. The tool compensates for this by taking a lazy approach to compilation, where a minimal, but sufficient, representation of the full state machine is passed to the state exploration engine. The state machine is then generated by the state exploration engine as it verifies (this is

[27]Of course, latency is rather a non-issue for totally asynchronous networks – as it is simply nondeterministic as to when, if ever, a message is delivered to the intended recipient.

[28]We assume here that the node processes are topology aware.

[29]This is one change which is likely to result in a state-space reduction.

akin to the 'on-the-fly' model-checking approach of the SPIN tool). Modeling techniques exist to maximise this lazy approach to compilation [54, 53]. Some of these techniques, such as the use of renaming, and the 'lazy spy', are generally applicable techniques. Others are more opportunistic and application-specific - such as the fact that the contract-signing protocol happens to have a particularly nice *Deductions* function that admits a highly optimised $SYNC$ model as described in Section 4.2.6.

These state-space problems are present when modelling any message-based communication in the face of an adversary. Each message sent by a correct party over the network needs to be buffered by the $SYNC$ process so as to be forwarded, after some fixed latency, to the intended recipient. (In the case of an asynchronous communication network, possibly after some non-deterministic latency.) Any message sent over the network is potentially 'overheard' by the adversary, and, therefore, must be regarded as data that could be used in some subsequent attack. The upshot of this is that our model of the adversary (over synchronous or asynchronous networks) must remember what messages have been sent 'up to now'. So, if up to n messages are potentially sent by correct parties, then, in the most pessimistic scenario, the $SYNC$'s buffer could be in any one of 2^n different states. In one of the contract-signing models, 115 different messages might, if we did not know better, be sent between the signatories during the course of a contract-signing session. On the face of it, this necessitates a $SYNC$ model of no less than 2^{115} states. No automated finite-state modeling tool could feasibly process a model this size, and so we are faced with the problem of substantially reducing the message-space without weakening the verification.

The generally accepted approach to this problem is to curtail the 'universal' message-space, Msg, as much as possible. Then it is argued 'off-line' that the verification is not weakened in any of the reduced message-space abstractions. Usually the argument is along the lines that the correct nodes would, anyway, only be processing a smaller, recognizable, and correctly typed subset of some superset of messages. Any messages that might be received outside of this smaller, 'expected', message-set can be, and are, ignored by correct nodes. One has to be careful that any such off-line arguments do not unintentionally eliminate some unforeseen trace of events, which could lead to a security breach. Therefore, it is better if we write our models and refinement tests in such a way that FDR, itself, does as much of the work as possible – conducting an exhaustive check on our behalf for all possible anomalous behaviour.

One possible solution to this dilemma entails incorporating in our model what is sometimes referred to as a 'watchdog'. The term 'watchdog' is broadly used, but, generally, it refers to any modeling device that is used to flag an error or warning if there is any divergence from some 'assumed' behaviour. The behavioural 'assumptions' will, more often than not, be exploited in some way in the model (e.g., to reduce state-space). For example, we might have to model our network plus adversary so that it only processes messages that we presuppose, from our reading of the protocol descriptions, will be sent by the correct nodes over the *send* channel, and in the order that we predict that they will be sent in. However, if we were wrong in our supposition, and a correct node did, at some point, attempt to send a message out of order, then we must raise a special *error* event flag.[30] We then have only to check that our final model is literally *error*-free in the traces model to assert that the order we predicted the messages would be sent was in fact correct.[31]

For the synchronous contract-signing model, we were able to split the verification into six test scripts, each referring to no more than four correct parties (those being two signatories plus the TTP and the Verifier). The number of messages sent by the correct parties during the course of a contract-signing session totalled no more than about ten, resulting in a manageable $SYNC$ buffer of some 2^{10} states.

If we look ahead to future verifications of protocols, in which correct parties may send upwards of $n = 30$ messages in total, being able to predict the sequence in which those n messages will be sent could be critical. It may mean the difference between failure, and being able to achieve a model that is analysable using FDR.[32] Perhaps, though, it is not too optimistic to expect such predictable communications when working with deterministic, reactive protocols (whether they are running over synchronous or asynchronous networks).

4.4.4 Liveness Properties and Denial-of-Service Attacks

A 'liveness' property is a property that asserts that some, usually desirable, event must happen at some point, as opposed to what are sometimes referred to as 'safety' properties which, generally, rule out certain undesirable behaviours. A liveness property that has been of particular concern to us was that of 'termination' – the specifying that a protocol will eventually terminate (successfully or not) after a bounded or unbounded period of time.

Consider, for example, the termination requirement on synchronous contract signing: that a correct signatory, A, eventually outputs either $out.A.signed.t.c$ or $out.A.failed.t.c$ sometime subsequent to a user instruction, $input.A.sign.B.c.t$, to sign a contract, c, with signatory B, the session's transaction identifier being t. This requirement was carefully worded in the original protocol documentation, and was intended as one of a generic set of requirements of contract-signing protocol implementations. As such, it would have been inappropriate for the authors to have stipulated a maximum number of rounds before termination was required, but only that particular implementations should terminate, and terminate in a pre-determined number of rounds.

Now, it is an interesting, and rather non-intuitive, facet of the CSP algebra that al-

[30]We would have to code into the model the supposed sequence in which each correct node sends its messages, and, also, keep count of the number of messages sent thus far. If a correct node were to, unexpectedly, send a message out of order, then the *error* event is engaged in.

[31]An assertion that would otherwise have to be argued off-line.

[32]We would hope to realize the system in a model with state-space that was polynomial in n, rather than exponential in n. This theory, however, is yet to be tested in practice. It may be that it is only pragmatic as part of a so-called 'W-SPEC' optimisation. W-SPEC is new research being undertaken by DERA under the auspices of the [19] EU project.

though it is quite trivial to write a generic CSP_M specification to the effect that either $out.A.signed.t.c$ or $out.A.failed.t.c$ will 'eventually happen', it is not possible to write a compilable, i.e., finite-state, CSP_M process that actually exhibits such behaviour (i.e., that refines the generic specification).[33] We qualify this. It is possible to write processes that will engage in $out.A.signed.t.c$ or $out.A.failed.t.c$ deterministically before, say, nine *tock*s, but not one that guarantees to engage in $out.A.signed.t.c$ or $out.A.failed.t.c$ after some non-deterministic number of *tock*s.[34]

This was a very particular concern when modeling the asynchronous contract-signing protocol where 'eventual' termination was wholly dependent upon our being able to abstract from the communication network 'eventually' delivering all messages.

For the synchronous contract-signing model, it was sufficient to prove that termination took no more than seven rounds. Proving this, however, was still problematic. For example, one perfectly legitimate set of traces is $\{\langle send.F.A.unsigned\_message, receive.A.F.unsigned\_message\rangle^n | n \in \mathbb{N}\}$ – corresponding to the adversary repeatedly sending some unsigned message to correct party A, which is not given a chance to do anything more than repeatedly receive that message (and discard it).

Because all possible behaviours are considered by FDR in a refinement test, potential denial-of-service attacks, such as this, noticeably prevent the protocol from terminating. It was a minor point of contention as to whether the original protocol description did, in fact, explicitly discount such attacks – although that they were being discounted (implicitly) was quite clear from the context.

Rather than weaken the fault/adversary model, and, therefore, the legitimacy of our verification, we can formulate a refinement test that simply discounts such denial-of-service attacks, as, indeed, does the original protocol specification. This is quite straightforward. The specification process is parametrized by a variable, n, corresponding to the round number by which termination was required, its alphabet, $\alpha(SPEC)$, say, is $\{tock, out.A.signed.t.c, out.A.failed.t.c\}$. The specification is as follows. Initially only $tock$ events are allowed. If and when a user inputs a request to sign contract c with transaction identifier t, then at most n more *tock*s can occur before at least one of $out.A.signed.t.c$ or $out.A.failed.t.c$ is engaged in, after which the specification process behaves like $RUN(\alpha(SPEC))$. We then hide all events in the alphabet of the synchronous contract-signing process with the exception of $\alpha(SPEC)$, and then check that this process both traces-refines the specification process and is deadlock-free. The deadlock freedom test is necessary to ensure that to do nothing from some point on is not an option – the

[33]The specification is simply $(out.A.signed.t.c \rightarrow DIV)[](out.A.failed.t.c \rightarrow DIV)$. A process, P, with alphabet α will eventually engage in either $out.A.signed.t.c$ or $out.A.failed.t.c$ if and only if $P\backslash\alpha - \{out.A.failed.t.c, out.A.signed.t.c\}$ is a failures-divergences refinement of the specification.

[34]The problem comes down to the lack of 'fairness' in CSP. Basically, whenever there is a choice between two events, a and b, say, it is always possible for a to be chosen over b, no matter how many times the choice is made. A 'fair arbiter' can be implemented in CSP, but it requires an infinite number of states [53].

contract-signing process could diverge any time, but, such behaviour aside, it must perform the visible traces of the specification.

This solution is, perhaps, not as intuitively convincing as we would have liked, but it is certainly preferably to artificially inhibiting adversarial or faulty behaviour (which we would then need to argue did not weaken the validation) that would be free to occur in practice. It also has the advantage of being relatively economical – requiring little effort in formulating a succinct specification process.

Leuschel et al. [34] have showed how certain classes of LTL formulae, using, for example, the eventual operator, $\diamond$, can be formulated – albeit with some difficulty – as refinement checks in a CSP setting. We considered formalising our 'eventual termination' specifications along the lines of this theory. It required additional refinement conditions which effectively allow for all those potential cases (traces) in which those events that we did want to 'eventually happen' do not appear (and so can be safely discarded). However, it soon became apparent that the resulting, augmented, specifications would be very unwieldy and unlikely to be compilable by FDR, and so this approach was abandoned. This type of verification, however, will be critical when verifying lifeness properties over asynchronous networks.

4.4.5 Typing Ambiguities in Messages

A message format can be said to be 'strongly-typed' if it is impossible for an honest principal to be deceived into interpreting the fields of a message in that format in ways other than intended by an honest sender. For instance, at the data level, a key and an identifier may conceivably be bit streams of similar length. In a poorly typed message, an adversary may deceive an honest principal into equating an agent identifier with part of a key. Consequently, poorly typed messages will admit a far greater range of possible attacks.

In general, it is possible to study the security implications of *particular* types of ambiguity by re-defining the deductions (inference rules) of the hostile agent. In a similar vein, we may study algebraic properties of encryption schemes.

For the contract-signing protocols, the message payload type is abstracted to the point that the question of whether it is strongly typed is trivially answered in the affirmative. This is justifiable though, because the original protocol specification itself assumes a-priori a strongly-typed message format. However, this may not always be so, and potential typing ambiguities may need to be studied in more detail in future.

4.5 Conclusions and Future Work on CSP

We have described a generic synchronous communications network with adversarial behaviour. This is the first model in the library of MAFTIA techniques which we will develop. The CSP analysis successfully identified where the specification was imprecise (which could lead to a security flaw). Further, after appropriate corrections were made, it was possible to obtain positive verifications for all requirements.

We have completed a second version of the synchronous contract signing model that adheres strictly to the original protocol description (without any generalizations) and the formal synchrony model. This required no changes to the $SYNC\_WITH\_SPYA, B, m_0$ model beyond changing the *latency* to 1 (so that *tocks* correspond to the formal model's end-of-round).

Managing compilation and state-explosion proved to be a challenge, but were not insurmountable. The abstraction arguments used and presented in this report were particular to the case study under consideration. However, we believe that these abstractions may not necessarily be protocol-specific, but may be applicable to more general synchronous protocols. This was discussed in section 4.4.3 and will be developed further. Future work will also involve investigating other generic MAFTIA techniques for addressing such problems, also to be included in the MAFTIA techniques library.

Modeling of the asynchronous contract-signing protocol is almost complete. Modeling asynchrony is considerably easier than modeling synchrony. Asynchrony doesn't require any guarantees on message delivery (i.e. that messages are received by the intended recipient within the specified latency period), and can be modelled easily using non-determinism. An asynchronous version of the contract-signing $SYNC\_WITH\_SPYA, B, m_0$ is much easier to formulate, and surprisingly compact. The results of this work will be published in a later deliverable.

The synchronous contract-signing protocol runs over an invariable, simple fully-connected communications network, in which all links have the same latency. However, it is very likely that in future verifications of protocols running over different synchrony models, such as partially synchronous networks, it will be necessary to consider more complex network topologies and differing link latencies. In a TTCB environment, for example, one has an essentially asynchronous 'payload' network, plus a reliable 'control' network. Future work will see the completion of the asynchrony models, and address the modelling of partial-synchrony, and other important MAFTIA middleware components.

5 Conclusions and the Way Forward

We have presented a rigorous model of basic concepts of MAFTIA by giving a new, general definition of the security of reactive systems using a simulatability approach. It comprises various types of faults (attacks) and topology as considered in MAFTIA (and also synchrony, but this was not shown here). A proof of secure message transmission, a low-level middleware protocol, was shown in this model, another proof of a more complicated protocol is given in [45].

In addition to this work on Objective 1, we have taken large steps towards Objectives 2 and 3 of this workpackage, the specification and verification of MAFTIA concepts using CSP, (i.e. in a formal language and with a model checker), and on a sound combination of cryptographic and formal analysis techniques. The CSP verification of protocols requires that suitable abstractions be made to make the protocols under consideration amenable to model checking techniques. This is an approach common in all forms of formal verification, theorem proving and model checking. However, such abstractions require that the assumptions on which they are based also be proven (not always as obvious as those made in the CSP analysis of the contract signing protocol). This requires hand proof, not necessarily a negative requirement since hand proofs often tease out implicit assumptions upon which security could be crucially dependent. The goal of the verification component of this work package is to make verification of MAFTIA concepts as automatic as is possible, and in conjunction, to reduce the necessary non-automated proof component to being as simple as possible.

In addition to finalizing the asynchronous version of the general rigorous model presented in Section 2, work is under way on a special abstract model of several cryptographic primitives. The faithfulness of this abstract model will be rigorously proven – once and for all – by hand. We hope to then bring the CSP network model closer to the rigorous model, so that the ideal systems and abstract compositions of the rigorous model can be implemented in CSP with the least possible effort. The end result will be a faithful linking of the cryptographic and the formal world in such a way that enables tool-support, or even automatic proofs of cryptographic protocols.

Bibliography

[1] M. Abadi, *Protection in Programming-Language Translations*, 25th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 1443, Springer-Verlag, Berlin 1998, 868–883

[2] M. Abadi, A. D. Gordon, *A Calculus for Cryptographic Protocols: The Spi Calculus*, 4th Conf. on Computer and Communications Security, ACM, 1997, 36–47

[3] M. Abadi, R. Needham, *Prudent Engineering Practice for Cryptographic Protocols*, IEEE Transactions on Software Engineering 22/1 (1996), 6–15

[4] M. Abadi, P. Rogaway, *Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)*, IFIP TCS 2000, LNCS 1872, Springer-Verlag, 2000, 3–22

[5] N. Asokan, Matthias Schunter, Michael Waidner, *Optimistic Protocols for Fair Exchange*, 4th ACM Conference on Computer and Communications Security, Zürich, April 1997, 6–17

[6] P. Baran, *On Distributed Communications: IX. Security, Secrecy, and Tamper-Free Considerations*, Memorandum RM-3765-PR, August 1964, The Rand Corporation. Reprinted in: Lance J. Hoffman (ed.): Security and Privacy in Computer Systems; Melville Publishing Company, Los Angeles, 1973, 99–123

[7] D. Beaver, *Secure Multiparty Protocols and Zero Knowledge Proof Systems Tolerating a Faulty Minority*, J. of Cryptology 4/2 (1991), 75–122

[8] M. Bellare, R. Canetti, H. Krawczyk, *A modular approach to the design and analysis of authentication and key exchange protocols*, 13th Symp. on Theory of Computing (STOC), ACM, 1998, 419–428

[9] M. Bellare, A. Desai, D. Pointcheval, P. Rogaway, *Relations Among Notions of Security for Public-Key Encryption Schemes*, Crypto '98, LNCS 1462, Springer-Verlag, 1998, 26–45

[10] M. Bellare, Ph. Rogaway, *Entity Authentication and Key Distribution*, Crypto '93, LNCS 773, Springer-Verlag, 1994, 232–249

[11] M. Ben-Or, O. Goldreich, S. Micali, R. L. Rivest, *A Fair Protocol for Signing Contracts*, IEEE Transactions on Information Theory 36/1 (1990), 40–46

[12] M. Blum, *Three Applications of the Oblivious Transfer*, Version 2: September 18, 1981, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkley, Ca. 94720

[13] R. Canetti, *Security and Composition of Multiparty Cryptographic Protocols*, J. of Cryptology 13/1 (2000) 143–202 (preliminary version in Theory of Cryptography Library 98-18, 1998)

[14] R. Canetti, *A unified framework for analyzing security of protocols*, Cryptology ePrint Archive, Report 2000/067, `http://eprint.iacr.org/`, December 2000

[15] R. Canetti, S. Goldwasser, *An Efficient Threshold Public Key Cryptosystem Secure Against Adaptive Chosen Ciphertext Attack*, Eurocrypt '99, LNCS 1592, Springer-Verlag, 1999, 90–106

[16] R. Cramer, V. Shoup, *A Practical Public Key Cryptosystem Provably Secure Against Adaptive Chosen Ciphertext Attack*, Crypto '98, LNCS 1462, Springer-Verlag, 1998, 13–25

[17] I. B. Damgård, *Collision free hash functions and public key signature schemes*, Eurocrypt '87, LNCS 304, Springer-Verlag, Berlin 1988, 203–216

[18] W. Diffie, M. E. Hellman, *New Directions in Cryptography*, IEEE Transactions on Information Theory 22/6 (1976), 644–654

[19] *Dependable Systems of Systems*, EU funded project

[20] D. Dolev, A. C. Yao, *On the Security of Public Key Protocols*, IEEE Transactions on Information Theory 29/2 (1983) 198–208

[21] Formal Systems (Europe) Ltd., *Failures-Divergences Refinement*, `http://www.formal.demon.co.uk/`

[22] R. Gennaro, S. Micali, *Verifiable Secret Sharing as Secure Computation*, Eurocrypt '95, LNCS 921, Springer-Verlag, 1995, 168–182

[23] O. Goldreich, *Secure Multi-Party Computation*, Working Draft, Version 1.1, September 21, 1998, available from `http://www.wisdom.weizmann.ac.il/users/oded/pp.htm`

[24] O. Goldreich, *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness*, Algorithms and Combinatorics 17, Springer-Verlag, Berlin 1999

[25] S. Goldwasser, L. Levin, *Fair Computation of General Functions in Presence of Immoral Majority*, Crypto '90, LNCS 537, Springer-Verlag, 1991, 77–93

[26] S. Goldwasser, S. Micali, *Probabilistic Encryption*, J. of Computer and System Sciences 28 (1984), 270–299

[27] S. Goldwasser, S. Micali, R. L. Rivest, *A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks*, SIAM Journal on Computing 17/2 (1988), 281–308

[28] S. Goldwasser, S. Micali, C. Rackoff, *The Knowledge Complexity of Interactive Proof Systems*, SIAM J. on Computing 18/1 (1989), 186–207

[29] O. Goldreich, S. Micali, A. Wigderson, *How to play any mental game—or—a completeness theorem for protocols with honest majority*, 19th Symp. on Theory of Computing (STOC), ACM, 1987, 218–229

[30] M. Hirt, U. Maurer, *Player Simulation and General Adversary Structures in Perfect Multiparty Computation*, J. of Cryptology 13/1 (2000), 31–60

[31] *MAFTIA-Project homepage*, `http://www.newcastle.research.ec.org/maftia/`, EU funded project

[32] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall 1985

[33] P. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, Crypto '96, LNCS 1109, Springer-Verlag, 1996, 104–113

[34] M. Leuschel, T. Massart, A. Currie, *How to make FDR spin: LTL model checking of CSP by refinement*, Technical Report DSSE-TR-2000-10, Department of Electronics and Computer Science, University of Southampton, September 2000. To appear in Proceedings of FME 2001

[35] P. Lincoln, J. Mitchell, M. Mitchell, A. Scedrov, *A Probabilistic Poly-Time Framework for Protocol Analysis*, 5th Conf. on Computer and Communications Security, ACM, 1998, 112–121

[36] P. Lincoln, J. Mitchell, M. Mitchell, A. Scedrov, *Probabilistic Polynomial-Time Equivalence and Security Analysis*, Formal Methods (FM'99), LNCS 1708, Springer-Verlag, 1999, Vol I, 776–793

[37] G. Lowe, *Breaking and fixing the Needham-Schroeder public-key protocol using FDR*, Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1055, Springer-Verlag, 1996, 147–166

[38] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, San Francisco 1996

[39] C. Meadows, *Using Narrowing in the Analysis of Key Management Protocols*, Symp. on Security and Privacy, IEEE, 1989, 138–147

[40] S. Micali, P. Rogaway, *Secure Computation*, Crypto '91, LNCS 576, Springer-Verlag, 1992, 392–404

[41] J. K. Millen, *The Interrogator: A Tool for Cryptographic Protocol Security*, Symp. on Security and Privacy, IEEE, 1984, 134–141

[42] B. Pfitzmann, *Sorting Out Signature Schemes*, 1st Conf. on Computer and Communications Security, ACM, 1993, 74–85

[43] B. Pfitzmann, M. Schunter, M. Waidner, *Optimal Efficiency of Optimistic Contract Signing*, 17th Symposium on Principles of Distributed Computing (PODC), ACM, New York 1998, 113–122

[44] B. Pfitzmann, M. Schunter, M. Waidner, *Secure Reactive Systems*, IBM Research Report RZ 3206 (#93252), IBM Research Division, Zürich, May 2000

[45] B. Pfitzmann, M. Schunter, M. Waidner, *Provably Secure Certified Mail*, IBM Research Report RZ 3207 (#93253), IBM Research Division, Zürich, Aug. 2000

[46] B. Pfitzmann, M. Schunter, M. Waidner, *Cryptographic Security of Reactive Systems*, Electronic Notes in Theoretical Computer Science (ENTCS) 32 (2000), available at `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/menu.htm`

[47] B. Pfitzmann, M. Waidner, *A General Framework for Formal Notions of "Secure" System*, Hildesheimer Informatik-Berichte 11/94, Universität Hildesheim, April 1994, available at `http://www.semper.org/sirene/lit/abstr94.html#PfWa_94`

[48] B. Pfitzmann, M. Waidner, *Composition and Integrity Preservation of Secure Reactive Systems*, 7th Conference on Computer and Communications Security, ACM, New York 2000, 245–254

[49] B. Pfitzmann, M. Waidner, *A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission*, IEEE Symposium on Security and Privacy, Oakland, May 2001

Preliminary version: IBM Research Report RZ 3304 (#93350), IBM Research Division, Zürich, Dec. 2000, and Cryptology ePrint Archive, Report 2000/066, `http://eprint.iacr.org/`, Dec. 2000

[50] M. O. Rabin, *Transaction Protection by Beacons*, Journal of Computer and System Sciences 27/ (1983), 256–267

[51] Ch. Rackoff, D. R. Simon, *Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack*, Crypto '91, LNCS 576, Springer-Verlag, Berlin 1992, 433–444

[52] A. W. Roscoe, *Modelling and Verifying Key-Exchange Protocols Using CSP and FDR*, 8th Computer Security Foundations Workshop, IEEE, 1995, 98–107

[53] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall 1998

[54] P. Y. A. Ryan, S. A. Schneider et al, *The Modelling and Analaysis of Security Protocols*, Addison Wesley

[55] M. Schunter, *Optimistic Fair Exchange*, Ph.D. Thesis, Technische Fakultät der Universität des Saarlandes, Saarbrücken, 2000

[56] V. Shoup, *On Formal Models for Secure Key Exchange*, IBM Research Report RZ 3076 (#93122), IBM Research Division, Zürich, November 1998. Also as Theory of Cryptography Library 99-12, last revised November 1999, `http://philby.ucsd.edu/cryptolib/`

[57] A. C. Yao, *Protocols for Secure Computations*, 23rd Symp. on Foundations of Computer Science (FOCS), IEEE, 1982, 160–164

[58] A. C. Yao, *Theory and Applications of Trapdoor Functions*, 23rd Symp. on Foundations of Computer Science (FOCS), IEEE, 1982, 80–91