Cryptographically Sound Analysis of Security Protocols

Dissertation zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Eingereicht von Michael Backes

Gutachter:
Prof. Dr. Birgit Pfitzmann
Prof. Dr. Harald Ganzinger

Dekan:
Prof. Dr. Philipp Slusallek

Kolloquium:

Saarbrücken, April 2002

Abstract

In this thesis, we show how formal methods can be used for the cryptographically sound verification of concrete implementations of security protocols in order to obtain trustworthy and meaningful proofs, and to eliminate human inaccuracies.

First, we show how to derive secure concrete implementations of a given abstract specification. The security proofs are essentially based on the well-established approach of bisimulation which can be formally verified yielding rigorous proofs. As an example, we present both a specification and a secure implementation of secure message transmission with ordered channels. Moreover, the example comprises a general methodology how secure implementation of arbitrary specifications can be obtained.

Thereafter, we concentrate on the actual goals the protocol should fulfill. Thus, we define integrity properties in our underlying model and we show that logic derivations among them carry over from the specification to the concrete implementation, which makes them accessible for tool-assisted verification. As an example, we formally verify one concrete protocol using the theorem prover PVS yielding the first machine-aided and sound proof of a cryptographic protocol.

As additional properties of security protocols, we consider liveness and non-interference. The standard definition of these properties is not suited to cope with protocols involving real cryptographic primitives, so we introduce new definitions which are restricted to polynomial runs and include error probabilities. We show that both properties carry over from the specification to the concrete implementation, and we present two examples, one for each property, which we prove to fulfill our definitions.

Kurzzusammenfassung

Diese Arbeit behandelt formale Verifikation von Sicherheitsprotokollen mit dem Ziel, maschinell verifizierte Beweise zu ermöglichen, die die kryptographische Semantik respektieren, d.h., deren Aussagen bzgl. der zugrundeliegenden Kryptographie und den kryptographischen Sicherheitsdefinitionen gültig sind (engl. *cryptographically sound proofs*).

Als erstes zeigen wir, wie formale Methoden benutzt werden können, um sichere konkrete Implementationen anhand einer gegebenen abstrakten Spezifikation herzuleiten. Wir geben dafür eine allgemeingültige Methologie an, die auf formal verifizierten Bisimulationen basiert, was uns rigorose und glaubhafte Sicherheitsbeweise liefert. Als Beispiel geben wir eine Spezifikation und eine konkrete Implementation für sichere geordnete Nachrichtenübertragung an. Die im Sicherheitsbeispiel der Implementation auftretende Bisimulation verifizieren wir mit Hilfe des Theorembeweisers PVS.

Als zweites konzentrieren wir uns auf die Ziele, die ein Sicherheitsprotokoll erfüllen soll. Wir definieren Integritätseigenschaften in unserem zugrundeliegenden Modell, und wir beweisen, dass sich logische Schlussfolgerungen bzgl. dieser Eigenschaften von der Spezifikation auf die Implementation übertragen, was eine essentielle Voraussetzung für maschinelle Verifikation darstellt. Als Beispiel verifizieren wir ein konkretes Protokoll mit Hilfe des Theorembeweisers PVS, was uns den ersten Beweis eines Sicherheitsprotokolls liefert, der sowohl maschinell verifiziert ist als auch der kryptographischen Semantik “treu” bleibt, d.h., der wirklich ein Beweis gegen die kryptographischen Primitive und deren kryptographische Sicherheitsdefinitionen ist.

Als zusätzliche Eigenschaften von Sicherheitsprotokollen betrachten wir Lebendigkeit (engl. *liveness*) und Unbeeinflussbarkeit (engl. *non-interference*). Da sich die Standarddefinition dieser wichtigen Eigenschaften als ungeeignet für echte Kryptographie herausstellt, führen wir allgemeinere Definitionen ein, die auf polynomielle Länge beschränkt sind und Fehlerwahrscheinlichkeiten berücksichtigen. Wir zeigen, dass sich diese Eigenschaften von der Spezifikation auf die Implementation übertragen, was wiederum den Bezug zu formalen Methoden herstellt. Wir präsentieren zwei Beispiele, je eines für jede Eigenschaft, von denen wir beweisen, dass sie die entsprechende Definition erfüllen.

Zusammenfassung

Kryptographische Protokolle besitzen in unserer heutigen Zeit einen zunehmend gr¨oßeren Stellenwert. Viele von ihnen werden bereits t¨aglich von Millionen von Menschen anhand des Internets benutzt, wie z.B. Online-Banking oder Protokolle zum sicheren Austausch von Geld und Ware. Offensichtlich ist die Sicherheit solcher Protokolle eine zentrale Anforderung, um von den Menschen akzeptiert und benutzt zu werden. Leider beruht die Sicherheit vieler Protokolle lediglich auf der Tatsache, das bis jetzt kein Angriff darauf gefunden wurde, bzw. alle bis jetzt gefundenen Sicherheitsl¨ucken wurden ausgemerzt. Die stetig steigenden Verluste von Banken durch Hackerangriffe zeigen deutlich, dass ein solcher Ansatz keinerlei wirkliche Sicherheit bietet. Protokolle brauchen einen *Beweis* ihrer Sicherheit, um akzeptiert werden zu k¨onnen. Desweiteren sollte ein solcher Beweis maschinell verifiziert sein, um menschliche Fehler weitestgehend auszuschließen.

Obwohl das Problem der formalen Verifikation von Protokollen bereits seit l¨angerem von vielen Wissenschaftlern behandelt wird, konnten bis jetzt nur stark idealisierte Protokolle bewiesen werden, die keinerlei Entsprechung mehr in der realen Welt besaßen. Demzufolge sind die bewiesenen Sicherheitsaussagen zumindest von zweifelhafter Bedeutung f¨ur die reale Welt. In solchen stark idealisierten Protokollen wird sogenannte 'perfekte Kryptographie' angenommen—gem¨aß dem Ansatz von Dolev und Yao [17]—was den Bezug zur realen Welt zerst¨ort. Perfekte Kryptographie heißt im allgemeinen, dass kryptographische Operationen als Term-Algebra aufgefasst werden in der lediglich vordefinierte Gleichungen und K¨urzungsregeln gelten. Im Falle von z.B. asymmetrischer Verschl¨usselung betrachtet man zwei Operatoren E_X und D_X, die ein Schl¨usselpaar f¨ur einen Teilnehmer X repr¨asentieren sollen. Die Gleichung $D_X(E_X(t)) = t$ gilt per Definition f¨ur alle Terme t. Will man nun beweisen, dass ein betrachtetes Ereignis (z.B. "die Nachricht t geht irgendwann im Klartext ¨uber das Netz") nicht gelten kann, so berechnet man den Abschluss der Term-Algebra und zeigt, dass das Ereignis in diesem Abschluss nicht enthalten ist. Offensichtlich ist eine Art Vollst¨andigkeitsaussage n¨otig, um sinnvolle Ergebnisse zu erhalten, d.h., man will eine Aussage der Form, dass eine Gleichung nur gelten kann, wenn sie syntaktisch anhand der gegebenen Gleichungen hergeleitet werden kann.

Betrachtet man nun allerdings echte kryptographische Primitive und ihre Sicherheitsdefinitionen, so sieht man im Gegensatz zu obiger Aussage, dass kryptographische Sicherheitsdefinitionen keinerlei Aussagen ¨uber *alle* Gleichungen treffen. Die Definition der Sicherheit von asymmetrischer Verschl¨usselung bei aktiven Angriffen besagt beispielsweise nur, dass der Angreifer durch seine Angriffe keinerlei partielle Informationen ¨uber den Klartext herausbekommen kann. Allerdings ist es nicht verboten, dass er z.B. Beziehungen zwischen Schl¨usseltexten findet. Man kann Beispiele konstruieren, in denen Beweise die mit perfekter Kryptographie arbeiten, schiefgehen, obwohl die verwendeten kryptographischen Primitive sicher sind im kryptographischen Sinn [47].

Leider ist ein gewisses Maß an Abstraktion unvermeidbar, da der in der Kryptographie auftretende Probabilismus f¨ur formale Methoden (bis jetzt) nicht zug¨anglich ist. Allerdings werden wir in dieser Arbeit zeigen, dass man anstelle der perfekten Kryptographie sogenannte 'treue' Abstraktionen betrachten kann, die den Bezug zur Kryptographie nicht zerst¨oren und trotzdem maschinelle Verifikation erm¨oglichen. Alles in allem stellen wir in dieser Arbeit eine allgemeine Methologie vor, wie maschinell verifizierte Beweise f¨ur konkrete Protokolle der realen Welt durchgef¨uhrt werden k¨onnen, so dass die Semantik der Kryptographie erhalten bleibt, d.h. der Beweis ist g¨ultig bzgl.

der zugrundeliegenden kryptographischen Primitive und deren kryptographischen Sicherheitsdefinitionen.

Überblick und Ergebnisse

Bevor wir uns in dieser Arbeit der eigentlichen Verifikation zuwenden, f¨uhren wir unser formales Modell f¨ur asynchrone reaktive Systeme ein. Das Modell selbst wurde von Pfitzmann und Waidner entwickelt und in [49] erstmalig publiziert, und ist demzufolge kein Verdienst dieser Arbeit.

Im Anschluss daran betrachten wir einige Modellvarianten, die wir als ¨aquivalent zum Standardmodell beweisen, was den Aufwand k¨unftiger Beweise erheblich reduziert, da wir immer die Variante annehmen k¨onnen, die f¨ur das betrachtete Problem am geeignetsten ist. Danach befassen wir uns kurz mit dem synchronen Modell von Pfitzmann, Schunter und Waidner [47], das als Vorg¨anger des hier betrachteten asynchronen Modells angesehen werden kann. Da viele Protokolle in der Praxis synchron, d.h. in Runden ablaufen, ist dieses Modell immer noch essentiell f¨ur viele praxisrelevante Protokolle. Demzufolge wollen wir zweigleisig fahren, aber ohne jedes Theorem und jedes Lemma f¨ur beide Modelle zu beweisen. Dieses Problem k¨onnte umgangen werden, indem man zeigt, dass es sich bei dem synchronen Modell lediglich um einen Spezialfall des asynchronen handelt, der nicht gesondert betrachtet werden muss. Diese Arbeit vollzieht den ersten, essentiellen Schritt dieses Ansatzes: Wir zeigen dass sich die Menge der synchronen System in die Menge der asynchrone Systeme einbetten lassen, so dass die Beziehung zwischen Spezifikation und Implementation erhalten bleibt, d.h. dass sich Sicherheitseigenschaften der asynchronen Einbettung auf das synchrone Modell ¨ubertragen.

Nach diesen modellspezifischen Beweisen wendet wir uns der eigentlichen maschinellen Verifikation von Sicherheitsprotokollen zu. Wir werden maschinelle Verifikation zu zwei unterschiedlichen Zielen verwenden:

Als erstes zeigen wir, wie formale Methoden benutzt werden k¨onnen, um zu einer gegebenen abstrakten Spezifikation eine sichere Implementierung herzuleiten. Der Beweis der Sicherheit der Implementierung umfasst eine allgemeine Methologie wie Protokollverifikation in Zukunft aussehen k¨onnte (und unserer Meinung nach auch sollte). Die Methologie basiert auf formal verifizierten Bisimulationen, was uns rigorose und glaubhafte Beweise liefert. Exemplarisch stellen wir eine abstrakte Spezifikation und eine konkrete Implementation von geordneter sicherer Nachrichten¨ubertragung vor. Den dabei auftretenden Bisimulationsbeweis haben wir formal mit Hilfe des Theorembeweisers PVS [44] verifiziert.

Als zweites betrachten wir die Ziele, die ein Protokoll erf¨ullen soll. Wir zeigen wie sich Integrit¨atseigenschaften in unserem zugrundeliegenden Modell ausdr¨ucken lassen, und wir zeigen, dass sich Integrit¨atseigenschaften von abstrakten Spezifikationen auf die konkreten Implementationen ¨ubertragen. Desweiteren zeigen wir, dass logische Schlussfolgerungen bzgl. Integrit¨atseigenschaften f¨ur die konkrete Implementierung g¨ultig sind im kryptographischen Sinn. Diese beiden Punkte erm¨oglichen es uns, die abstrakte Spezifikation anstelle der konkreten Implemantation maschinell zu verifizieren. Der Vorteil ist, dass Spezifikationen im allgemeinen deterministisch und somit zug¨anglich f¨ur formale Beweissysteme sind, w¨ahrend der Probabilismus der konkreten Implementationen f¨ur formale Beweissysteme (bis jetzt) noch nicht zug¨anglich ist. Der Beweis der Spezifikation ¨ubertr¨agt sich dann automatisch auf die konkrete Implementierung. Exemplarisch verifizieren wir unsere abstrakte Spezifikation f¨ur geordnete sichere Nachrichten¨ubertragung mit Hilfe des Theorembeweiser PVS, d.h. wir bewei-

sen, dass eine Umordnung der Nachrichten oder ein Replayangriff von unserem System verhindert wird.

Als weitere Ziele betrachten wir Fairness, Liveness und Non-Interference (Unbeeinflussbarkeit). Wir werden sehen, dass die Standarddefinitionen dieser wichtigen Eigenschaften ungeeignet f¨ur echte Kryptographie sind, da sie auf unendlichen Programmabl¨aufen beruhen, w¨ahrend kryptographische Protokolle nach polynomiell, also endlich, vielen Schritten abbrechen. Desweiteren ber¨ucksichtigen unsere Definitionen Fehlerwahrscheinlichkeiten, was einen wichtigen Bezug zur Kryptographie herstellt. Analog zu Integrit¨atseigenschaften zeigen wir, dass sich diese Eigenschaften von der abstrakten Spezifikation auf die konkrete Implementation ¨ubertragen. Um unsere Liveness-Definition anhand eines realen Systems zu illustrieren, geben wir eine Spezifikation und eine Implementation f¨ur Nachrichten¨ubertragung mit verf¨ugbaren Kan¨alen an, und wir zeigen, dass beide unsere Definition erf¨ullen. Um unsere Non-Interference-Definition zu veranschaulichen, geben wir eine kryptographische Firewall an, die zwei ehrliche Benutzer sicher miteinander kommunizieren l¨asst und sie vom Rest der Welt abschirmt, d.h. sie k¨onnen nicht von anderen Benutzeren oder vom Angreifer beeinflusst werden. Die Firewall l¨asst sich problemlos auf mehrere, disjunkte Parteien erweitern, was dem typischen Konzept mehrerer Firewalls im Internet entspricht.

Acknowlegdements

First of all, I am profoundly grateful to my main adviser Birgit Pfitzmann, who gave me the opportunity to work in security and cryptography, and proposed the topic of this work. Whenever I encountered a problem which I had problems to solve on my own, she took the time to sit together until we finally found a solution. Without her ongoing support this work would have undoubtedly impossible.

I thank my co-adviser Harald Ganzinger for his interest in my work and for several valuable comments which helped to increase the quality of the thesis.

I thank J¨org Siekmann for general advice in several non-technical questions which arose during the last year.

I'm indebted to Christian Jacobi for being a qualified co-author of the papers which form the tool-assisted basis of this thesis. Moreover, he suffered for me and proof-read several parts of this work. Thanks a lot! I would like to thank Michael Steiner and Michael Waidner for producing new ideas and useful comments about polynomial fairness and liveness; furthermore, I thank Heiko Mantel for many valuable comments about non-interference.

I thank the former and current members of our group at Saarland University: Ammar Alkassar, Alexander Geraldy, André Adelsbach, Ahmad-Reza Sadeghi, Chris St¨uble, Matthias Schunter, Michael Steiner, Petra Maschke, and Sandra Steinbrecher for providing an inspiring environment which helped me a lot finishing this work. In particular, I thank Sandra Steinbrecher for being a congenial room mate who had to bear with me during the last year.

This work was supported by the Graduate Studies Program "Quality Guarantees for Computer Systems" founded by the DFG. Thanks a lot, mainly for the financial support :-)

Last, but certainly not least, I am grateful to my parents who always supported me. I am grateful to my girlfriend Isabell Schu for a wonderful last year, and to all my friends, especially Swen Jacobs, Sabine Schwierczek, David Mendzigall, Christian Marbert, and Dirk Leinenbach who always had to be appreciative of my overdrawn eagerness to finish this work, and who reminded me that there are much more important things than university.

Contents

Chapter 1

Introduction and Overview

Nowadays, cryptographic protocols are getting more and more attention in both theory and practice. As common examples, we may think of online banking, fair exchange over the internet, or the even more sensitive topic of electronic elections. In the early days of security research, these protocols were designed using a simple iterative process: someone proposed a protocol, someone else found an attack, the bug was fixed, and so on, until no further attacks were found. Today, it is commonly accepted that this approach does not give any security guarantee at all, since many simple and apparently secure protocols have been found incorrect over the years. Moreover, important protocols like fair contract signing, electronic auctions or payments are just too complex for this approach. Secure protocols—or more generally, secure reactive systems—need a *proof* of security before being acceptable.

Some years ago, this field of research only focused on certain cryptographic primitives such as encryption and digital signature schemes. In current research, larger systems like secure channels or fair exchange protocols are to be verified. The main goal researchers are ultimately aiming at is to verify really large systems like whole e-commerce architectures.

If we turn our attention to what has already been done, we can distinguish between two main approaches that unfortunately seem to be rather disjoint. One approach mainly considers the cryptographic aspects of protocols aiming at complete and mathematically rigorous proofs with respect to cryptographic definitions. The other one involves formal methods, so protocols should be verified using formal proof systems, or these proofs should even be generated automatically by theorem provers. Usually, these proofs are much trustworthier than hand-made proofs, especially if we consider large protocols using many single steps. The main problem of this approach lies in the necessary abstraction of cryptographic details. This abstraction cannot be completely avoided, since formal methods cannot handle probabilistic behaviours by now, so usually perfect cryptography is assumed—following the approach of Dolev and Yao [17]—in order to make machine-aided verification possible.[1] As we will see, these abstractions are unfaithful in the sense that they cannot be securely implemented, i.e., there is no guarantee that a formally proven protocol is actually secure if implemented with a cryptographically secure primitive [47, 4].

Comparing both approaches, we can see that cryptographic proofs are more mean-

[1] At the moment, we are working on a probabilistic calculus which should be able to automatically deduct whether two given probabilistic machines are computationally bisimilar, i.e., whether they are bisimilar up to a very small error probability [6]. This may help to avoid some of those abstractions.

ingful in the sense of security but they also have one main disadvantage: cryptographic proofs are usually very long, sketchy, and error-prone even for very small examples like encryption schemes, and moreover have to be done by hand so far. Hence, it seems rather impossible to verify whole e-commerce architectures by now.

Our goal is to link both approaches to get the best overall result: proofs that allow abstraction and the use of formal methods, but retain a sound cryptographic semantics. Being more precise, we want to formally verify abstract goals of abstract protocols using formal proof tools, and we want these proofs to carry over to the concrete goals of the concrete protocols, enabling both trustworthy *and* cryptographically sound proofs of arbitrary protocols. We will now describe our approach in more detail while addressing the problems of common verification techniques.

Abstract 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. In order to make this possible, abstractions must get rid of probabilism yielding the already mentioned notion of perfect cryptography. In order to achieve this, cryptographic operations are 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 [17]. For instance, there are a two operators E_X and D_X for asymmetric en- and decryption representing a key pair of a participant X. Twofold encryption of a message m from a basic message space M does 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 to hold, 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 with respect to the considered issue and formalism, e.g., [41, 39]; nowadays most work is based on standard languages and general-purpose verification tools, as initiated, e.g., in [52, 31, 2].

Unfortunately, these models lack a link between the chosen abstractions and the real cryptographic primitives as defined and proven in cryptography. The main problem of these models is not even that one somehow has to weaken the statements to polynomial-time adversaries and allow error probabilities; the problem is that proofs based on formal methods prove a property to hold by showing that the negation of the property is not contained in the closure of the term algebra. Obviously, there has to be some kind of completeness in order to obtain meaningful results. In contrast to that, the definitions in cryptography 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, 14]), but nothing is asserted about possible relations on the ciphertexts. One can construct examples, at least artificial ones, where proofs made with these abstractions go wrong with encryption schemes provably secure in the cryptographic sense [47].

Faithful Abstractions

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 which are easier to fulfill by actual cryptography. Our approach belongs to the second way. Both approaches presuppose that one defines what it means that some abstraction is implemented in a cryptographic sense. Both also need proofs

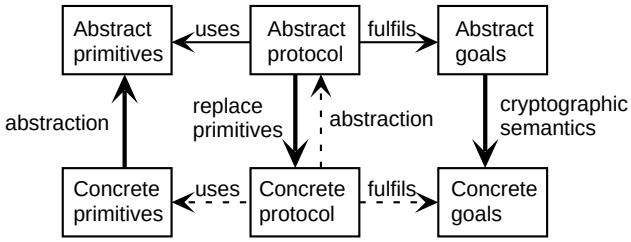

Figure 1.1: Goals of faithful abstraction. Bold arrows should be defined once and for all, normal arrows once per protocol. It should be proven that dashed arrows follow automatically.

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.

The lower layer of the figure represents the real world, involving concrete systems using concrete cryptographic primitives. The task is to show that they fulfill certain concrete goals. Our approach now starts by defining what faithful abstractions of concrete primitives in fact are. This corresponds to the left part of the figure, and has already been defined in [49] using the concept of simulatability modern cryptography often uses. These abstractions should provide non-probabilistic, deterministic interfaces hiding all cryptographic details, hence they can be formally validated using existing formal methods. Now, we would like to obtain an abstraction of the overall protocol by suitably combining the abstract primitives. In [49], it has already been shown that this abstraction in fact retains the soundness of the system. Moreover, it introduces composition of systems and it contains a composition theorem which states that a protocol can be refined step-wise by replacing the abstract primitives yielding a concrete protocol again which uses concrete primitives. This corresponds to the middle part of Figure 1.1. The final task of the approach is to relate abstract and concrete goals of the overall protocol, and to show that fulfillment in the abstract system implies fulfillment in the concrete system with respect to the cryptographic semantics, which is shown in this thesis. This corresponds to the right side of Figure 1.1.

Thus, all bold arrows in Figure 1.1 have been defined and all dashed arrows have been proven. Now, abstract goals of abstract protocols can be formally verified by formal proof systems and the proofs automatically carry over to the concrete goals of the concrete system, enabling both trustworthy *and* cryptographically sound proofs of arbitrary protocols.

Although the task of finding these abstract primitives and proving them to be faithful turns out to be labour-intensive, it has to be done only *once* for every considered primitive. Since common protocols are usually based on only very few, basic primitives like secure channels and commitments, the work is well spent. By now, only faithful abstractions of secure message transmission [49], fair exchange protocols [56], and secure group key exchange [59] have been shown. We will review the scheme for secure message transmission in Section 2.3, since it serves as the foundation for many examples in this work. We only briefly state here that there is current work at our group at Saarland University and IBM Zurich on designing a cryptographic library involving additional primitives, nonces, timestamps and so on.

Moreover, we will see that abstract specifications usually are monolithic, i.e., they consist of only *one* machine, so they have to be split in proper parts before they can

eventually be refined. This corresponds to the upper-left arrow of Figure 1.1, i.e., we have to show which abstract primitives are used by the abstract protocol. We will show in this work that formal methods are well-suited for formally proving that the split system has the same functional behaviour as the original, monolithic system using the well-established approach of bisimulation. In combination with the composition theorem, these formally verified bisimulations are a powerful linkage between concrete systems and their abstract counterparts.

The Actual Goals

As actual goals of a cryptographic protocol, we consider integrity properties, fairness and liveness, and absence of information flow in this work. At first, we concentrate on integrity properties, and we prove that they in fact carry over to the concrete implementation. Moreover, we show that logic derivations among them are valid for the implementation in the cryptographic sense, which makes them accessible for theorem provers. After that, we focus on the common concept of fairness and liveness. Unfortunately, the standard definition of fairness and liveness is not suited for most cryptographic protocols since it is based on infinite runs whereas the runs of such protocols are restricted to polynomial length, i.e., they are finite. In order to circumvent this problem and to cope with real cryptography, we introduce the notion of *polynomial fairness* and *polynomial liveness* which handles the restriction of the runs to polynomial length and moreover allows error probabilities. Similar to integrity properties, we show that liveness properties carry over to the concrete implementation under some reasonable assumptions (we do not have to show this for fairness, since fairness is a property of a scheduler, not of a system). Finally, we show how to express the very complex approach of information flow using the well-established concept of probabilistic non-interference. The common definitions of probabilistic non-interference do not include computational details like error probabilities which would be essential to cope with systems involving real cryptography, so we introduce the notion of *computational probabilistic non-interference*. Just as we did with the preceeding properties, we show that non-interference properties carry over to the concrete implementation.

Related Literature

Verification of cryptographic protocols is a large field of research, and lots of things still have to be done. As already stated above, one main goal is to retain a sound cryptographic semantics and nevertheless apply machine-aided verification in order to obtain formally verified proofs. This goal is pursued by several researchers: our approach is based on the model of Pfitzmann and Waidner recently introduced in [49], which we believe to be really close to this goal. Other possible ways to achieve this goal have been presented in [29, 30]: actual cryptography and security is directly expressed and verified using a formal language (π-calculus), but their approach does neither offer any abstractions nor abstract interfaces that enable tool support. [33] has quite a similar motivation to our underlying model, but it is restricted to the usual equational specifications (following the approach of [17], the so-called “Dolev-Yao model”), and the semantics is not probabilistic. Moreover, [33] only considers passive adversaries and only one class of users exists, which is referred to as “environment”. So the abstraction from cryptography is no more faithful than abstraction in other formal methods papers about security, e.g., [31, 55, 3, 45, 18]. They are all based on intuitive, but unfaithful

abstractions, i.e., they cannot be securely implemented, since no cryptographic scheme fulfills their requirements.

Thus, there has not been any proof of correctness of a cryptographic protocol so far which both includes formal proof systems and retains the soundness of cryptography. This thesis bridges this gap by giving the first example of a machine-aided but nevertheless sound proof of an integrity property of a concrete cryptographic protocol. Moreover, the example comprises a general methodology how protocol verification might be performed in the future.

For the sake of readability we postpone the related literature of fairness, liveness, and non-interference to their corresponding chapters.

Organization of the Thesis

Chapter 2 contains a detailed review of the reactive model of Pfitzmann and Waidner for asynchronous systems, originally presented in [49], which we will use throughout the thesis. We believe their model to be well-suited for analyzing cryptographic protocols; moreover, we participated in the further development of the model during the last year, so we presume to use the term "our model" in this work. The chapter concludes with a review of the abstract and concrete system of secure message transmission [49] which serves as a building block for designing larger examples in this work.

Starting with Chapter 3, we present original work.

Chapter 3 discusses several model variants, especially focusing on the relations between four different kinds of simulatability. We show that all four definitions are equivalent. This significantly reduces our effort in further proofs since we can always use the definition which is suited best for the considered problem. The second part of the chapter is dedicated to the relationship of our underlying asynchronous model and its synchronous predecessor [47]. In practice, lots of protocols are synchronous, i.e., they proceed in rounds. As commonly known examples, the reader may think of smart cards, fair exchange protocols over the internet, etc. Thus, the synchronous model is still essential to cope with protocols currently used in practice; hence, we want to drive double tracked, but without proving each and every theorem for both models. A possibility to circumvent this problem is to show that the synchronous model can be regarded as a special case of the asynchronous model, which we do not have to consider separately. This work contains the first, essential step of this task: we show that synchronous systems can be embedded into asynchronous ones such that simulatability is preserved by this embedding. This result serves as the foundation for carrying over lemmas and theorems from the asynchronous case to the synchronous case.

Chapter 4 deals with the actual verification of cryptographic protocols. We present a monolithic specification of secure message transmission with ordered channels and a concrete, secure implementation. The way of actually deriving the implementation comprises a general methodology how concrete implementations of abstract specifications can be found. Moreover, the methodology contains formally verified bisimulations, which yields trustworthy proofs. The bisimulation that occurred in the security proof of our example is formally verified using the theorem prover PVS [44]. Prior to this work, there has not been any success in using the advantage of formal verification in order to derive cryptographically sound implementations, so our methodology is new, and it serves as our first step of bridging the gap between the rigorous proofs of

cryptography and verification using formal proof systems.

Chapter 5 finishes building this bridge. We define what it means for a system to provide integrity properties in a cryptographic sense. We then show that proofs of such properties made for the abstract specification also hold for the concrete implementation, and that logic derivations among integrity properties are valid for the implementation in the cryptographic sense, which makes them available for theorem provers. We conclude with the formal verification of our specification of the previous section, i.e., we show that message reordering is in fact prevented. We again use PVS to obtain a formally verified proof. According to our results, the proof automatically carries over to the concrete implementation which yields the first machine-aided, but nevertheless sound proof of the concrete goals of a concrete system. Together with the result of the previous chapter, this completes the first tool-supported and cryptographically sound proof of both the security of a concrete implementation and its actual goals.

The remaining chapters show that the important concepts of fairness, liveness, and even the very complex notion of information flow are comprised in our model.

In Chapter 6 we show how fairness and liveness can be expressed in our model. We will see that the standard definitions cannot be applied for most cryptographic protocols, so we introduce the more general notion of *polynomial fairness* and *polynomial liveness*, which makes both concepts accessible for arbitrary real cryptographic protocols for the first time. As usual, we show that these properties carry over from the abstract system to its concrete counterpart, and we present an example which is polynomially live with respect to a desired property.

In Chapter 7 we show that information flow can be expressed in our model using the well-established concept of probabilistic non-interference. We introduce the more general notion of *computational probabilistic non-interference* which is essential to cope with real cryptography. Similar to the previous chapter, we show that non-interference properties of the abstract specification carry over to the concrete implementation. As a practical example, we present a specification and a secure implementation of a cryptographic firewall guarding two honest users from their environment. Moreover, the specification can easily be generalized to multiple disjoint parties which comprise arbitrary numbers of users.

Chapter 8 summarizes and gives an outlook to future research.

Chapter 2

Asynchronous Reactive Systems

In this chapter, we introduce our formal model of reactive systems in asynchronous networks that we will use throughout the thesis. It should be noted once more that this is not original work, but only a review of the model recently introduced by Pfitzmann and Waidner [49]. However, we have made some minor, but far-reaching changes since then. For example, explicit master schedulers are now considered which allow to achieve well-known properties like liveness or non-interference. In the original model, the adversary was always forced to be the master scheduler, corresponding to an intuitive, but nevertheless restricting assumption. Moreover, we extended their channel model to reliable, non-authenticated channels which turned out to be essential for proving typical examples of non-interference, e.g., a cryptographic firewall guarding several honest users from their environment. Finally, we helped to strengthen the model, i.e., to find and correct errors.

For the sake of completeness, we review the definitions in full detail. We additionally use informal descriptions of the definitions in order to illustrate how the actual intuition of expressing a model for sound protocol verification has been put into formulas step by step.

2.1 General System Model

In the following we consider a finite alphabet Σ. The set of all strings over Σ will be denoted by Σ^*, ϵ denotes the empty word and $\Sigma^+ := \Sigma^* \setminus \{\epsilon\}$. Moreover, we consider some special symbols $!, ?, {}^{\leftrightarrow}, {}^{\triangleleft} \notin \Sigma$ that will be used to express different ports of machines. For $s \in \Sigma^*$ and $l \in \mathbb{N}_0$, we define $s\lceil_l$ to be the l-bit prefix of s.

Our machine model is probabilistic state-transition machines, similar to probabilistic I/O automata as sketched by Lynch [32]. Communication between different machines is done over ports which can be divided into input and output ports. Inspired by the CSP-Notation [26] we write output and input ports as $\mathsf{p}!$ and $\mathsf{p}?$, respectively. Formally, ports are defined as follows.

Definition 2.1 *(Ports)*

a) A port p *is a triple* $(\mathsf{q}, l, d) \in \Sigma^+ \times \{\epsilon, {}^{\leftrightarrow}, {}^{\triangleleft}\} \times \{!, ?\}$. *We call* $\mathsf{name}(p) := \mathsf{q}$ *its* name, $\mathsf{label}(p) := l$ *its* label, *and* $\mathsf{dir}(p) := d$ *its* direction. *In the following these triples are simply written as concatenations, i.e., we write* $\mathsf{q}^{\triangleleft}!$ *instead of* $(\mathsf{q}, {}^{\triangleleft}, !)$ *and so on.*

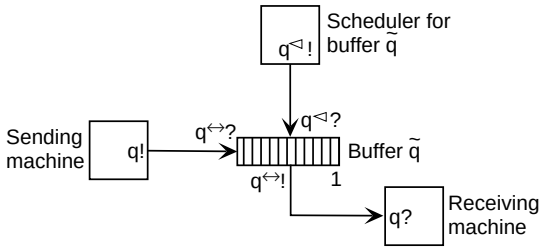

Figure 2.1: Ports and buffers.

b) We call a port (q, l, d) *an* input port *or* output port *iff* $d = ?$ *or* $d = !$*, respectively. We call it a* simple port, buffer port *or* clock port *iff* $l = \epsilon, \leftrightarrow$*, or* $\triangleleft$*, respectively. For a set* P *of ports let* $\mathsf{out}(P) := \{p \in P \mid \mathsf{dir}(p) = !\}$ *and* $\mathsf{in}(P) := \{p \in P \mid \mathsf{dir}(p) = ?\}$*. We use the same notation for sequences of ports, retaining the order.*

c) By p^c*, the (*low-level*)* complement *of a port* p*, we denote the port with which it connects according to Figure 2.1, i.e.,* $\mathsf{q}^{\triangleleft}!^c := \mathsf{q}^{\triangleleft}?$*,* $\mathsf{q}!^c := \mathsf{q}^{\leftrightarrow}?$*,* $\mathsf{q}^{\leftrightarrow}!^c := \mathsf{q}?$*, and vice versa. Accordingly, we define the (low-level) complement of a set or sequence of ports.*

d) For a simple port p*, we define its* high-level complement p^C *as the port connected to* p *without counting the buffer, i.e.,* $\mathsf{q}!^C := \mathsf{q}?$ *and vice versa.*

◇

Ports will later be connected by naming convention, i.e., a port $\mathsf{p}!$ always sends messages to $\mathsf{p}?$.

After introducing ports, we can now focus on machines. If a machine is switched, it receives an input tuple at its input ports and performs its transition function yielding a new state and an output tuple in the deterministic case, or a finite distribution over the set of states and possible outputs in the probabilistic case. At each switching step of one particular machine, at most one value can arrive at every input port and the machine can at most produce one output per port. Furthermore, each machine has a bound on the length of the considered inputs which allows time bounds independent of the environment.

Definition 2.2 *(Machines) A* machine *is a tuple*

$$\mathsf{M} = (name_{\mathsf{M}}, Ports_{\mathsf{M}}, States_{\mathsf{M}}, \delta_{\mathsf{M}}, l_{\mathsf{M}}, Ini_{\mathsf{M}}, Fin_{\mathsf{M}})$$

of a name $name_{\mathsf{M}} \in \Sigma^+$*, a finite sequence* $Ports_{\mathsf{M}}$ *of pairwise distinct ports, a set* $States_{\mathsf{M}} \subseteq \Sigma^*$ *of states, a probabilistic state-transition function* δ_{M}*, a length function* $l_{\mathsf{M}} : States_{\mathsf{M}} \to (\mathbb{N} \cup \{\infty\})^{|\mathsf{in}(Ports_{\mathsf{M}})|}$*, and sets* $Ini_{\mathsf{M}}, Fin_{\mathsf{M}} \subseteq States_{\mathsf{M}}$ *of initial and final states. Its input set is* $\mathcal{I}_{\mathsf{M}} := (\Sigma^*)^{|\mathsf{in}(Ports_{\mathsf{M}})|}$*; the* i*-th element of an input tuple denotes the input at the* i*-th input port. Its output set is* $\mathcal{O}_{\mathsf{M}} := (\Sigma^*)^{|\mathsf{out}(Ports_{\mathsf{M}})|}$*. The empty word,* ϵ*, denotes no in- or output at a port.* δ_{M} *maps each pair* $(s, I) \in States_{\mathsf{M}} \times \mathcal{I}_{\mathsf{M}}$ *to a finite distribution over* $States_{\mathsf{M}} \times \mathcal{O}_{\mathsf{M}}$*. If* $s \in Fin_{\mathsf{M}}$ *or* $I = (\epsilon, \ldots, \epsilon)$*, then* $\delta_{\mathsf{M}}(s, I) = (s, (\epsilon, \ldots, \epsilon))$ *deterministically. Inputs are ignored beyond the length bounds, i.e.,* $\delta_{\mathsf{M}}(s, I) = \delta_{\mathsf{M}}(s, I\lceil_{l_{\mathsf{M}}(s)})$ *for all* $I \in \mathcal{I}_{\mathsf{M}}$*, where* $(I\lceil_{l_{\mathsf{M}}(s)})_i := I_i\lceil_{l_{\mathsf{M}}(s)_i}$ *for all* i*.* ◇

In the text, we often write "M" also for $name_{\mathsf{M}}$.

Machines usually start with one initial input, i.e., the starting state is parameterized. Complexity is measured in terms of the length of this initial input, often a security parameter k given in unary representation; in particular, polynomial-time is meant in this sense.

Note that the chosen representation makes δ_{M} independent of the port names. This will also hold for runs and views, cf. Definition 2.6 and 2.7. Hence, we can rename ports in some proofs without changing the views. The requirement for ϵ-inputs means that it does not matter if we switch a machine without inputs or not; we will also omit such steps from the runs. Inputs "masked" by a length bound 0 are treated in the same way. We call a machine M a *black-box submachine* of a machine M' if the machine M' has access to the state-transition function δ_{M} of M, i.e., it can execute δ_{M} for the current state of the machine and arbitrary inputs.[1]

Remark 2.1. In order to cope with specific inputs and outputs of a machine M, we introduce some additional notation which is not contained in the original model. Let P be a subset of the input ports of M, i.e., $P \subseteq \mathsf{in}(Ports_{\mathsf{M}})$, and $(v_i)_{i\in P} \in (\Sigma^+)^P$ be given. Then $\mathcal{I}_{\bigodot_{\mathsf{p}?\in P}\ \mathsf{p}?=v_{\mathsf{p}?}}$ denotes the input with $\mathsf{p}? = v_{\mathsf{p}?}$ for all $\mathsf{p}? \in P$ and $\mathsf{p}'? = \epsilon$ for all $\mathsf{p}'? \in \mathsf{in}(Ports_{\mathsf{M}}) \setminus P$. For the sake of readability, we do not explicitly define the set P in the following, i.e., we simply write $\mathcal{I}_{\mathsf{p}_1?=v_1,\ldots,\mathsf{p}_n?=v_n}$ in slight abuse of notation, instead of defining $P := \{\mathsf{p}_i? \mid i \in \{1,\ldots,n\}\}$. In the special case $\mathsf{p}_i? = \epsilon$ for all $\mathsf{p}_i? \in \mathsf{in}(Ports_{\mathsf{M}})$, i.e., in case of an all-empty input, we write $\mathcal{I}_\epsilon$. Outputs are defined similarly. ∘

The proposed machines have a natural realization as probabilistic turing machines if the state-transition function δ_{M} is computable, where each port is represented by one tape. Two machines that have a connected low-level input and output port share one tape for this channel which simply allows transmitting of arbitrary outputs.

Machines can be divided into three classes depending on the ports they use. The first two classes consist of the "usual" machines, they are dealt with in the following definition.

Definition 2.3 *(Simple Machines and Master Schedulers) A machine* M *is* simple *if it has only simple ports and clock out-ports. A machine* M *is a* master scheduler *if it has only simple ports and clock out-ports and the special* master-clock *in-port* $\mathsf{clk}^{\triangleleft}?$*. Without loss of generality, a master scheduler makes no outputs in a transition that enters a final state.* ◇

If we speak of machines in another context we usually consider simple machines. The special master scheduler is scheduled whenever a machine does not make any non-empty output at a clockout port or the scheduled buffer cannot deliver the requested message. Usually, the adversary is regarded as the master scheduler, but it is sometimes useful to define an explicit master scheduler to achieve certain goals like liveness and privacy properties.[2]

The third class consists of the already mentioned buffers. They will be inserted between every pair of high-level connected ports to ensure asynchronous behaviour.

[1] Sometimes, the machine M' is also allowed to "reset" the machine M, i.e., to take it back to a prior state. However, we omit it here since we do not need it in this work.

[2] The original model of [49] forces the adversary to be the master scheduler. Hence, the mentioned properties cannot be achieved there.

Roughly speaking, buffers store the messages in transit, i.e., they represent the net in the real world. A machine M can schedule the i-th stored message of buffer $\widetilde{\mathsf{p}}$ by sending i at $\mathsf{p}^{\triangleleft}!$ (provided that $\mathsf{p}^{\triangleleft}! \in Ports_{\mathsf{M}}$ holds). Formally, buffers are defined as follows.

Definition 2.4 *(Buffers) For each name $\mathsf{q} \in \Sigma^+$ we define a specific machine $\widetilde{\mathsf{q}}$, called a* buffer*: It has three ports, $\mathsf{q}^{\triangleleft}?$, $\mathsf{q}^{\leftrightarrow}?$, $\mathsf{q}^{\leftrightarrow}!$ (clock, in, and out) (see Figure 2.1). Its internal state is a queue over Σ^+ with random access, initially empty. Its set of final states is empty, and all its length bounds are infinite. For each state transition, if the input x at $\mathsf{q}^{\leftrightarrow}?$ is non-empty, then $\delta_{\widetilde{\mathsf{q}}}$ appends x to the queue. A non-empty input at $\mathsf{q}^{\triangleleft}?$ is interpreted as a number $i \in \mathbb{N}$ and the i-th element is retrieved (where 1 indicates the oldest one), removed from the queue, and output at $\mathsf{q}^{\leftrightarrow}!$. (This might be the element just appended.) If there are less than i elements, the output is ϵ.* ◇

By now, we only focused on single machines. We now consider finite sets $\hat{C}$ of machines with pairwise different machine names and disjoint sets of ports. We will call such a set a *collection*. Moreover, we define the *completion* $[\hat{C}]$ of $\hat{C}$ as the union of all machines of $\hat{C}$ and the buffers needed for every output port.

Definition 2.5 *(Collections)*

a) For every machine M, let $\mathsf{ports}(\mathsf{M})$ denote the set of ports in $Ports_{\mathsf{M}}$, and for a set $\hat{M}$ of machines, let $\mathsf{ports}(\hat{M}) := \bigcup_{\mathsf{M} \in \hat{M}} Ports_{\mathsf{M}}$.

b) A collection *$\hat{C}$ is a finite set of machines with pairwise different machine names, disjoint sets of ports, and where all machines are simple, master schedulers, or buffers. It is called* polynomial-time *if all its non-buffer machines have a polynomial-time implementation.*

c) Each set of low-level complementary ports $\{p, p^c\} \subseteq \mathsf{ports}(\hat{C})$ is called a low-level connection*, and the set of them the* low-level connection graph $\mathsf{gr}(\hat{C})$*. By $\mathsf{free}(\hat{C})$ we denote the* free *ports in this graph, i.e., $\mathsf{ports}(\hat{C}) \setminus \mathsf{ports}(\hat{C})^c$. A set of high-level complementary simple ports $\{p, p^C\} \subseteq \mathsf{ports}(\hat{C})$ is called a* high-level connection*, and the set of them the* high-level connection graph $\mathsf{Gr}(\hat{C})$*.*

d) A collection is closed *if $\mathsf{free}(\hat{C}) = \{\mathsf{clk}^{\triangleleft}?\}$. (Hence, there is exactly one master scheduler, identified by having the port $\mathsf{clk}^{\triangleleft}?$.)*

e) The completion *$[\hat{C}]$ of a collection $\hat{C}$ is the union of $\hat{C}$ and the corresponding buffer for each simple or clock out-port $q \in \mathsf{ports}(\hat{C})$.*

f) If $\widetilde{\mathsf{q}}, \mathsf{M} \in \hat{C}$ and $\mathsf{q}^{\triangleleft}! \in \mathsf{ports}(\mathsf{M})$ then we call M the scheduler for *buffer $\widetilde{\mathsf{q}}$ (in $\hat{C}$).*

◇

We have defined collections of machines, but we did not describe yet how they may interact, when they are scheduled and so on. Scheduling of machines is done sequentially, so we have exactly one active machine M at any time. This machine is allowed to schedule an arbitrary buffer $\widetilde{\mathsf{p}}$ for which $\mathsf{p}^{\triangleleft}! \in \mathsf{ports}(\mathsf{M})$. If $\widetilde{\mathsf{p}}$ is scheduled and it also can deliver the specified message, it implicitly schedules the receiving machine M', i.e., the unique machine with $\mathsf{p}? \in \mathsf{ports}(\mathsf{M}')$. If M tries to schedule multiple buffers at a time, only one is taken, and if no buffer is scheduled (or the scheduled

buffer cannot deliver the requested message) the master scheduler is scheduled. This yields the notion of runs (sometimes called *traces* or *executions*) of the collections. In order to ensure that runs are well-defined, we demand the collection to be closed, i.e., it especially must not have any free input ports. Since the machines themselves are probabilistic, we obtain a probability space of runs, along with their corresponding random variables.

Definition 2.6 *(Runs) Given a closed collection $\hat{C}$ with master scheduler X and a tuple $ini \in Ini_{\hat{C}} := \times_{\mathsf{M} \in \hat{C}} Ini_{\mathsf{M}}$ of initial states, the probability space of* runs *is defined inductively by the following algorithm. It has a variable r for the resulting run, an initially empty list, a variable $\mathsf{M_{CS}}$ ("current scheduler") over machine names, initially $\mathsf{M_{CS}} := \mathsf{X}$, and treats each port as a variable over Σ^*, initialized with ϵ except for $\mathsf{clk}^{\triangleleft}? := 1$. Probabilistic choices only occur in Phase (1).*

1. Switch current scheduler: *Switch machine $\mathsf{M_{CS}}$, i.e., set $(s', O) \leftarrow \delta_{\mathsf{M_{CS}}}(s, I)$ for its current state s and in-port values I. Then assign ϵ to all in-ports of $\mathsf{M_{CS}}$.*
2. Termination: *If X is in a final state, the run stops. (As X made no outputs, this only prevents repeated master clock inputs.)*
3. Buffer messages: *For each simple out-port $\mathsf{p}!$ of $\mathsf{M_{CS}}$, in their given order, switch buffer $\tilde{\mathsf{p}}$ with input $\mathsf{p}^{\leftrightarrow}? := \mathsf{p}!$. Then assign ϵ to all these ports $\mathsf{p}!$ and $\mathsf{p}^{\leftrightarrow}?$.*
4. Clean up scheduling: *If at least one clock out-port of $\mathsf{M_{CS}}$ has a value $\neq \epsilon$, let $\mathsf{q}^{\triangleleft}!$ denote the first such port and assign ϵ to the others. Otherwise, let $\mathsf{clk}^{\triangleleft}? := 1$ and $\mathsf{M_{CS}} := \mathsf{X}$ and go back to Phase (1).*
5. Scheduled message: *Switch $\tilde{\mathsf{q}}$ with input $\mathsf{q}^{\triangleleft}? := \mathsf{q}^{\triangleleft}!$, set $\mathsf{q}? := \mathsf{q}^{\leftrightarrow}!$ and then assign ϵ to all ports of $\tilde{\mathsf{q}}$ and to $\mathsf{q}^{\triangleleft}!$. Let $\mathsf{M_{CS}} := \mathsf{M}'$ for the unique machine M' with $\mathsf{q}? \in \mathsf{ports}(\mathsf{M}')$. Go back to Phase (1).*

Whenever a machine (this may be a buffer or even a black-box submachine) with name $name_{\mathsf{M}}$ is switched from (s, I) to (s', O), we add a step *$(name_{\mathsf{M}}, s, I', s', O)$ to the run r where $I' := I\lceil_{I_{\mathsf{M}}(s)}$, except if s is final or $I' = (\epsilon, \ldots, \epsilon)$. This gives a family of random variables*

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

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

Most of the time, we will not be interested in the whole run but only in its restriction to a set of machines, i.e., to the honest user. This restriction is called the *view* of these machines in this particular run.

Definition 2.7 *(Views) The* view *of a subset $\hat{M}$ of a closed collection $\hat{C}$ in a run r is the restriction of r to $\hat{M}$, i.e., the subsequence of all steps $(name_{\mathsf{M}}, s, I, s', O)$ where $name_{\mathsf{M}}$ is the name of a machine $\mathsf{M} \in \hat{M}$. This gives a family of random variables*

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

and similarly for l-step prefixes. For a singleton $\hat{M} = \{\mathsf{H}\}$ we write $view_{\hat{C}}(\mathsf{H})$ instead of $view_{\hat{C}}(\{\mathsf{H}\})$ for reasons of readability. ◇

2.1.1 Security-specific System Model

In this subsection we define specific collections for security purposes. We start by defining the actual system part; afterwards we focus on honest users and adversaries, which are usually referred to as the environment.

A *structure* is a pair $(\hat{M}, S)$, where $\hat{M}$ is a collection of simple machines, and $S \subseteq \mathsf{free}([\hat{M}])$, the so-called *specified ports*, are a subset of the free ports of $[\hat{M}]$. Specified ports always guarantee certain services like "send message m to A" for a message transmission system, or "transfer amount x to B" in a payment system. The users may only connect to a subset of these ports, the remaining free ports of the structure are additionally available to the adversary. We will always describe specified ports by their complements S^c, i.e., the ports honest users should have, because that is independent of the buffer notation. If we consider a set of structures we obtain a *system* Sys.

Definition 2.8 *(Structures and Systems)*

a) A structure *is a pair* $struc = (\hat{M}, S)$ *where* $\hat{M}$ *is a collection of simple machines called* correct machines*, and* $S \subseteq \mathsf{free}([\hat{M}])$ *is called* specified ports*. If* $\hat{M}$ *is clear from the context, let* $\bar{S} := \mathsf{free}([\hat{M}]) \setminus S$*. We call* $\mathsf{forb}(\hat{M}, S) := \mathsf{ports}(\hat{M}) \cup \bar{S}^c$ *the* forbidden ports*.*

b) A system Sys *is a set of structures. It is polynomial-time iff all its collections* $\hat{M}$ *are polynomial-time.*

◇

Note that we do not demand that the adversary is forbidden to connect to ports of S. However, a limitation of the adversary to the free ports of $\mathsf{forb}(\hat{M}, S)$ would not really be a restriction. We will show this in Section 3.2.

We will later consider cryptographic systems that are described using an intended structure for the case without attacks. The remaining structures are derived using a trust model for both machines and channels. As an example, consider the powerset of the set of machines as the desired trust model, so we have one structure for every possible combination of correct and incorrect machines (i.e., of honest and dishonest users).

A structure of a system can be completed to a *configuration* by adding additional machines H and A, modeling honest users and the adversary, respectively. The machine H is restricted to the specified ports S, A connects to the remaining free ports of the structure and both machines can interact, e.g., in order to model active attacks.

Definition 2.9 *(Configurations)*

a) A configuration *of a system* Sys *is a tuple* $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A})$ *where* $(\hat{M}, S) \in Sys$ *is a structure,* H *is a simple machine without forbidden ports, i.e.,* $\mathsf{ports}(\mathsf{H}) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$*, and the completion* $\hat{C} := [\hat{M} \cup \{\mathsf{H}, \mathsf{A}\}]$ *is a closed collection. The set of configurations is written* $\mathsf{Conf}(Sys)$*.*

b) The initial state of each machine in a configuration is a common security parameter k *in unary representation. This means that we consider the families of runs and views of the collection* $\hat{C}$ *restricted to the subset* $Ini'_{\hat{C}} := \{(1^k)_{\mathsf{M} \in \hat{C}} | k \in \mathbb{N}\}$ *of* $Ini_{\hat{C}}$*. We write* run_{conf} *and* $view_{conf}(\hat{M})$ *for the families* $run_{\hat{C}}$ *and* $view_{\hat{C}}(\hat{M})$ *restricted to* $Ini'_{\hat{C}}$*, and similar for* l*-step prefixes. Furthermore, we identify* $Ini'_{\hat{C}}$ *with* $\mathbb{N}$ *and thus write* $run_{conf,k}$ *etc. for the individual random variables.*

c) The set of configurations of Sys with polynomial-time user H and adversary A is called $\mathsf{Conf}_{\mathsf{poly}}(Sys)$. The index $_{\mathsf{poly}}$ is omitted if it is clear from the context.

◇

In the original model of [49], the adversary is forced to be the master scheduler of every configuration, which fits our intuition of asynchronous behaviour, i.e., the adversary should be able to schedule the network.

However, it will sometimes be useful to consider self-scheduled systems, i.e., systems containing an explicit master scheduler for each of its structures, which allows us to achieve goals like liveness and privacy properties, cf. Chapter 6 and 7.

2.1.2 Simulatability

The definition of security between two systems is based on the common concept of *simulatability*. Simulatability essentially means that whatever might happen to an honest user in an arbitrary configuration of a concrete system Sys_{real} can also happen in a configuration of an ideal system Sys_{id}.

Being more precise, for every configuration $conf_1 \in \mathsf{Conf}(Sys_{\mathsf{real}})$, there exists a configuration $conf_2 \in \mathsf{Conf}(Sys_{\mathsf{id}})$ yielding indistinguishable views of H in both configurations. We abbreviate this by $Sys_{\mathsf{real}} \geq_{\mathsf{sec}} Sys_{\mathsf{id}}$ and we say that Sys_{real} is "at least as secure" as the system Sys_{id}. A typical situation is illustrated in Figure 2.2. The notion of indistinguishability has been introduced in [65] and has asserted its position as a fundamental concept of modern cryptography. We will give a rigorous definition later on.

However, we do not want to compare a structure $(\hat{M}_1, S_1) \in Sys_{\mathsf{real}}$ with arbitrary structures of Sys_{id}, but only with certain "suitable" ones. What suitable actually means can be defined by a mapping f from Sys_{real} to the powerset of Sys_{id}, so that $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ means that $(\hat{M}_2, S_2)$ is such a suitable structure. The mapping f is called *valid* if the mapped structures always have the same set of specified ports.

Definition 2.10 *(Valid Mappings, Suitable Configurations) Let Sys_1 and Sys_2 be two systems.*

a) A valid mapping *for them is a function $f : Sys_1 \to \mathcal{P}(Sys_2)$ with $S_1 = S_2$ for all structures $(\hat{M}_1, S_1)$ and $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$.*

b) If Sys_2 contains exactly one structure $(\hat{M}_2, S_2)$ with $S_2 = S_1$ for each $(\hat{M}_1, S_1) \in Sys_1$, the canonical mapping *f is defined by $f(\hat{M}_1, S_1) = \{(\hat{M}_2, S_2)\}$.*

c) Given f, the set $\mathsf{Conf}^f(Sys_1)$ of suitable configurations *contains those configurations $(\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1)$ where $\mathsf{ports}(\mathsf{H}) \cap \mathsf{forb}(\hat{M}_2, S) = \emptyset$ for all $(\hat{M}_2, S) \in f(\hat{M}_1, S)$.*

◇

Since the definition of simulatability is based on indistinguishability of probability distributions, i.e., the views of H in both configurations have to be indistinguishable, we repeat the definition of indistinguishability essentially from Yao [65].

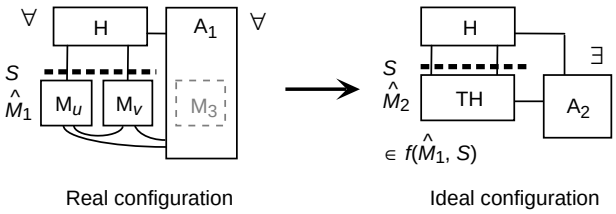

Figure 2.2: Example of simulatability. The view of H is compared.

Definition 2.11 *(Negligible Functions) A function $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is* negligible*, written $g(k) \leq 1/\mathsf{poly}(k)$, if for all positive polynomials Q, $\exists k_0 \forall k \geq k_0 : g(k) \leq 1/Q(k)$. The class of negligible functions is written $NEGL$.* ◇

Definition 2.12 *(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 are*

a) perfectly indistinguishable *("=") if for each k, the two distributions var_k and var'_k 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 functions bounded by $Q(k) \cdot 2^{-k}$ for a polynomial Q, and the (larger) class $NEGL$.

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, the distinguisher is given the security parameter and an element chosen according to either var_k or var'_k and he has to guess which distribution the element came from.)

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

We are now ready to introduce the simulatability definition.

Definition 2.13 *(Simulatability) Let systems Sys_1 and Sys_2 with a valid mapping f be given.*

a) We say $Sys_1 \geq^{f,\mathsf{perf}}_{\mathsf{sec}} Sys_2$ (perfectly at least as secure as*) if for every configuration $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^f(Sys_1)$, there exists a configuration $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ with $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ (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* $view_{conf_1,l}(\mathsf{H}) \approx_{SMALL} view_{conf_2,l}(\mathsf{H})$ *for all polynomials* l*, i.e., statistical indistinguishability of all families of* l*-step prefixes of the views.*

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.* ◇

2.1.3 Some Useful Lemmas

Sometimes we have to combine several machines into one machine that should provide the same functional behaviour as the collection of the original machines. We first introduce what "combination of several machines" actually means, and afterwards present a lemma which states that combinations are well-defined and moreover fit some necessary properties.

Definition 2.14 *(Combination of Machines) Let* $\hat{D}$ *be a collection without buffers. For a new name* $\mathsf{n_D}$ *(i.e., a name which is neither used as a machine name nor a port name in the considered collection so far), we define the* combination *of* $\hat{D}$ *into one machine* D *with this name, written* $\mathsf{comb}(\hat{D})$ *in slight abuse of notation.*

a) Its ports are $Ports_{\mathsf{D}} := \mathsf{ports}(\hat{D})$*. (Their order would be an additional parameter of* comb*, but it never matters in the following.)*

b) Its states are $States_{\mathsf{D}} := \times_{\mathsf{M}\in\hat{D}} States_{\mathsf{M}}$*.*

c) Its transition function δ_{D} *is defined by applying the transition function of each submachine to the corresponding substates and inputs, unless* D *has reached a final state (see below). In that case,* δ_{D} *does not change the state and produces no output.*

d) Its length function l_{D} *is defined by applying the length function of the corresponding submachine for each input port.*

e) Its initial states are $Ini_{\mathsf{D}} := \times_{\mathsf{M}\in\hat{D}} Ini_{\mathsf{M}}$*. For every* $k \in \mathbb{N}$*, we identify the state* $(1^k)_{\mathsf{M}\in\hat{D}}$ *with* 1^k *(for the conventions in configurations).*

f) If there is a master scheduler $\mathsf{X} \in \hat{D}$*, then* Fin_{D} *is the set of all states of* D *where* X *is in a state from* Fin_{X}*. Otherwise,* D *stops as soon as* all *submachines have stopped:* $Fin_{\mathsf{D}} := \times_{\mathsf{M}\in\hat{D}} Fin_{\mathsf{M}}$*.*

◇

Lemma 2.1 *(Combination) Let* $\hat{C}$ *be a collection without buffers,* $\hat{D} \subseteq \hat{C}$*, and* $\mathsf{D} := \mathsf{comb}(\hat{D})$ *with a name that is new in* $\hat{C}$*. Let* $\hat{C}^* := (\hat{C} \setminus \hat{D}) \cup \{\mathsf{D}\}$*.*

a) D *is well-defined.*

b) If $[\hat{C}]$ is a closed collection, then so is $[\hat{C}^*]$.

c) If $[\hat{C}]$ is closed then the view of any set of original machines in $[\hat{C}^*]$ is the same as in $[\hat{C}]$. This includes the views of the submachines in D, which are well-defined functions of the view of D.

d) Combination is associative: If $\hat{D} = \hat{D}_1 \cup \hat{D}_2$ and $\mathsf{D}_1 := \mathsf{comb}(\hat{D}_1)$, then $\mathsf{comb}(\{\mathsf{D}_1\} \cup \hat{D}_2) = \mathsf{D}$, if one identifies Cartesian products that differ only in the bracket structure.

e) If all machines in $\hat{D}$ are polynomial-time, then so is D.

□

Using our notion of combination we can introduce a special notion of simulatability called blackbox simulatability.

Definition 2.15 *(Universal and Blackbox Simulatability)* Universal simulatability *means that A_2 in Definition 2.13 does not depend on H (only on $\hat{M}_1$, S, and A_1).* Blackbox simulatability *means that A_2 is the combination of a fixed simulator Sim, depending at most on $\hat{M}_1$, S and $\mathsf{ports}(\mathsf{A}_1)$, and a machine A'_1 that differs from A_1 at most in the names and labels of some ports. The partial function σ that defines this renaming is tacitly assumed to be given with Sim. A_1 is then called a* blackbox submachine *of* Sim. ◇

Clearly, blackbox simulatability implies universal simulatability, and universal simulatability implies "standard" simulatability. We will finally state some lemmas that will be used throughout the thesis.

Lemma 2.2 *(Properties of Runs and Views) Let $\hat{C}$ be a closed collection.*

a) Whenever a machine M is switched in a run of $\hat{C}$, there is at most one port $p \in \mathsf{ports}(\hat{C})$ with $p \neq \epsilon$. If it exists, $p \in \mathsf{ports}(\mathsf{M})$.

b) Views of polynomial-time machines are always of polynomial size. If $\hat{C}$ is polynomial-time, the runs are of polynomial size.

□

Lemma 2.3 *(Valid Mappings and Suitable Configurations) Let two systems Sys_1 and Sys_2 with a valid mapping f be given.*

a) Then $S^c \cap \mathsf{forb}(\hat{M}_i, S) = \emptyset$ for $i = 1, 2$ for every $(\hat{M}_i, S) \in Sys_i$, i.e., the ports that users are intended to use are not at the same time forbidden (not even in the corresponding structures of the other system).

b) With regard to Sys_1 alone, the restriction to suitable configurations is without loss of generality in the following sense: For every $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1) \setminus \mathsf{Conf}^f(Sys_1)$, there is a configuration $conf_{\mathsf{f},1} = (\hat{M}_1, S, \mathsf{H}_\mathsf{f}, \mathsf{A}_{\mathsf{f},1}) \in \mathsf{Conf}^f(Sys_1)$ such that $view_{conf_{\mathsf{f},1}}(\mathsf{H}_\mathsf{f}) = view_{conf_1}(\mathsf{H})$.

□

The proof of this lemma is simply done by port renaming of the considered configuration. It can be found in [49].

Lemma 2.4 *(Indistinguishability)*

a) The statistical distance $\Delta(\phi(\mathsf{var}_k), \phi(\mathsf{var}'_k))$ of a function ϕ of 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 every function ϕ of them. The same holds for statistical indistinguishability with any class $SMALL$, and for computational indistinguishability if ϕ is polynomial-time computable and the elements of the domains D_k are of polynomial length in k.

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 facts are well-known, hence we omit the easy proof, cf. [49].

Lemma 2.5 *(Types of Security) If $Sys_1 \geq_{\mathsf{sec}}^{f,\mathsf{perf}} Sys_2$, then $Sys_1 \geq_{\mathsf{sec}}^{f,SMALL} Sys_2$ for every non-empty class $SMALL$. Similarly, $Sys_1 \geq_{\mathsf{sec}}^{f,SMALL} Sys_2$ for a class $SMALL \subseteq NEGL$ implies $Sys_1 \geq_{\mathsf{sec}}^{f,\mathsf{poly}} Sys_2$.* □

This lemma can be proven easily by applying Lemma 2.4 using that views of a polynomial-time machine always are of polynomial size, and that the distinguisher is a special case of the function ϕ.

What we finally want is the relation $\geq_{\mathsf{sec}}$ to be transitive which is captured by the following lemma.

Lemma 2.6 *(Transitivity) If $Sys_1 \geq^{f_1} Sys_2$ and $Sys_2 \geq^{f_2} Sys_3$, then $Sys_1 \geq^{f_3} Sys_3$, where $f_3 := f_2 \circ f_1$ is defined in a natural way as follows: $f_3(\hat{M}_1, S)$ is the union of the sets $f_2(\hat{M}_2, S)$ with $(\hat{M}_2, S) \in f_1(\hat{M}_1, S)$. This holds for perfect, statistical and computational security, and also for universal and blackbox simulatability.* □

The proof of this lemma is much more difficult than the previous ones. It can be found in detail in the original paper [49].

2.2 Special Cases and Composition

In this section we first introduce some special cases of the model that play an important role in security considerations. After that, we introduce composition of reactive systems and state the composition theorem, which allows us to refine a system step-wise by replacing the abstract primitives with their already proven cryptographic counterparts. The theorem enables modular proofs, so that systems can be designed completely idealized, and afterwards can be refined step by step.

As special classes of the model we consider structures in which there is exactly one machine for each user (which are all combined to the overall user H) and its machine is correct if and only if the user is honest. This corresponds to a usual real-life situation, because machines of honest users are usually considered correct, i.e., the user can regard them as trusted devices.

A main distinction of the introduced classes can be made between static and adaptive adversaries. Throughout the thesis we will only focus on static adversaries, i.e., we assume that it is a priori clear which users are faulty and which users are honest. In adaptive (or sometimes called dynamic) scenarios the set of corrupted machines may increase over time, e.g., because there is a special master adversary who can hack into machines in order to corrupt them [7, 12, 59]. Adaptive adversaries are more powerful than static ones, see [12] for an example that is secure in the static case but insecure against adaptive adversaries.

2.2.1 Standard Cryptographic Systems with Static Adversaries

As we already stated above we consider systems in which each honest user u controls exactly one machine M_u in every structure so that the machine works correct iff the user is honest. The system is derived using an intended structure $(\hat{M}^*, S^*)$ and a trust model. The intended structure corresponds to the case where every user is honest, the additional structures will be derived by leaving out incorrect machines (i.e., dishonest users) according to the considered trust model, i.e., they are considered as part of the adversary.

We define that all buffers that connect different machines are scheduled by the adversary unless explicitly mentioned otherwise. We only allow a machine M_u to schedule buffers that transport messages from the machine to itself, called a self-loop. We require all those connections to be secure. This allows us to define a machine M_u as a combination of local submachines. The case where the user in- and outputs are also treated in the same way is called localized.

Definition 2.16 *(Standard Cryptographic Structures and Trust Models) A* standard cryptographic structure *is a structure $(\hat{M}^*, S^*)$ where $\hat{M}^* = \{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$ with $n \in \mathbb{N}$ and $S^{*\,c} = \{\mathsf{in}_u!, \mathsf{out}_u?|u = 1, \ldots, n\}$, where $\mathsf{in}_u?$ and $\mathsf{out}_u!$ are ports of machine M_u. Each machine M_u is simple, and for all names p, if $p^{\triangleleft}! \in \mathsf{ports}(\mathsf{M}_u)$ then $p?, p! \in \mathsf{ports}(\mathsf{M}_u)$.*

A localized cryptographic structure *is the same except that for all $u = 1, \ldots, n$, $\mathsf{in}_u{}^{\triangleleft}!$ also belongs to $S^{*\,c}$ and $\mathsf{out}_u{}^{\triangleleft}!$ to $\mathsf{ports}(\mathsf{M}_u)$, but $\mathsf{out}_u? \notin \mathsf{ports}(\mathsf{M}_u)$.*

A standard trust model *for such a structure is a pair $(\mathcal{ACC}, \chi)$ of an* access structure *and a* channel model. *Here $\mathcal{ACC} \subseteq \mathcal{P}(\{1, \ldots, n\})$ is closed under insertion (of more elements) and denotes the possible sets of correct machines. χ is a mapping $\chi : \mathsf{Gr}(\hat{M}^*) \to \{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$. It characterizes each high-level connection as secure (private and authentic), authenticated (only authentic), or insecure (neither private nor authentic). If a connection c connects a machine M_u with itself, we require $\chi(c) = \mathsf{s}$.* ◇

Typical examples are threshold structures $\mathcal{ACC}_t := \{\mathcal{H} \subseteq \{1, \ldots, n\} \mid |\mathcal{H}| \geq t\}$ with $t \leq n$.

Definition 2.17 *(Standard Cryptographic Systems) Given a standard (or localized) cryptographic structure and trust model, the corresponding* standard (or localized) cryptographic system *is given by*

$$Sys_{\mathsf{real}} := \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) | \mathcal{H} \in \mathcal{ACC}\}$$

with $S^c_{\mathcal{H}} := \{\mathsf{in}_u!, \mathsf{out}_u? | u \in \mathcal{H}\}$, and $\mathsf{in}_u{}^{\triangleleft}!$ in the localized case, and $\hat{M}_{\mathcal{H}} := \{\mathsf{M}_{u,\mathcal{H}} | u \in \mathcal{H}\}$, where $\mathsf{M}_{u,\mathcal{H}}$ is derived from M_u as follows:

- *The ports* $\mathsf{in}_u?$ *and* $\mathsf{out}_u!$ *and all clock ports are unchanged.*
- *Consider a simple port* $p \in \mathsf{ports}(\mathsf{M}_u) \setminus \{\mathsf{in}_u?, \mathsf{out}_u!\}$*, where* $p^C \in \mathsf{ports}(\mathsf{M}_v)$ *with* $v \in \mathcal{H}$*, i.e.,* $c = \{p, p^C\}$ *is a high-level connection between two correct machines:*
  - *If* $\chi(c) = \mathsf{s}$ *(secure),* p *is unchanged.*
  - *If* $\chi(c) = \mathsf{a}$ *(authenticated) and* p *is an output port,* $\mathsf{M}_{u,\mathcal{H}}$ *gets an additional new port* p^d *(i.e., a port with a new name), where it duplicates the outputs at* p*. This can be done by a trivial blackbox construction. We assume without loss of generality that there is a systematic naming scheme for such new ports (e.g., appending* ^{d}*) that does not clash with prior names. The new 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, 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.*
- *Consider a simple port* $p \in \mathsf{ports}(\mathsf{M}_u) \setminus \{\mathsf{in}_u?, \mathsf{out}_u!\}$*, where* $p^C \notin \mathsf{ports}(\mathsf{M}_v)$ *for all* $v \in \mathcal{H}$*:*

  If p *is an output port, it is unchanged. If it is an input port, it is renamed into* p^a*. (In both cases the adversary can connect to it.)*

For localized systems, the same definition holds with the obvious modifications: the ports $\mathsf{in}_u{}^{\triangleleft}!$ *with* $u \in \mathcal{H}$ *also belong to* $S^c_{\mathcal{H}}$*, and* p *is only chosen in* $\mathsf{ports}(\mathsf{M}_u) \setminus \{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}!\}$. ◇

Obviously, all channel types presented above do not guarantee anything about the reliability of the channels. However, if we consider common concepts like non-interference or liveness properties, we need channels that *guarantee* that a message will eventually be scheduled. Moreover, it often fits our intuition to additionally make the channel accessible for the adversary which yields so-called *non-authenticated reliable* channels. However, we will postpone the formal definition to Chapter 6 where we present additional motivation for this channel type.

Definition 2.18 *(Standard Ideal Systems) A* standard (or localized) ideal system *is of the form*

$$Sys_{\mathsf{id}} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) | \mathcal{H} \in \mathcal{ACC}\}$$

for an access structure $\mathcal{ACC} \subseteq \{1, \ldots, n\}$ *for some* $n \in \mathbb{N}$ *and the same sets of specified ports as in corresponding real systems, i.e.,* $S^c_{\mathcal{H}} := \{\mathsf{in}_u!, \mathsf{out}_u?, (\mathsf{in}_u{}^{\triangleleft}!) | u \in \mathcal{H}\}$. ◇

One then compares a standard or localized real system with a standard or localized ideal system with the same access structure, using the canonical mapping (Definition 2.10).

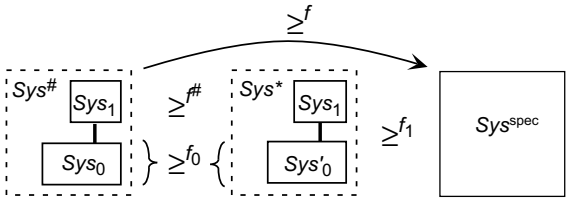

Figure 2.3: Overview over the composition theorem and its use in a modular proof. The left and middle part show the statement of Theorem 2.1, the right side is handled by Corollary 2.1.

2.2.2 Composition

In this section we introduce composition of reactive systems and state the composition theorem of [49]. We start outlining the basic idea. Assume that we have already proven a system Sys_0 to be at least as secure as another system Sys'_0 (typically an ideal system used as a specification), so we would like to use Sys_0 as a secure replacement for Sys'_0, i.e., we want to replace the specification with its implementation. Replacement means that we have another system Sys_1 that uses Sys'_0; we call this composition Sys^*, see Figure 2.3.

Inside of Sys^* we want to replace Sys'_0 with Sys_0 yielding a system $Sys^{\#}$. Usually, $Sys^{\#}$ is completely real whereas Sys^* is at least partly ideal. The situation is shown in the left and middle part of Figure 2.3. Intuitively, $Sys^{\#} \geq_{\mathsf{sec}} Sys^*$ should hold which captures the statement of the composition theorem.

So far, we did not examine the right side of the figure. As we already stated above Sys^* may be only partly ideal. However, system design usually starts with a completely abstract specification Sys^{spec} which fulfills the desired goals by construction. Moreover, such a specification is usually monolithic, so it has to be split in proper parts before it can eventually be refined using the composition theorem.

Corollary 2.1 will state that the real system $Sys^{\#}$ is at least as secure as the ideal specification Sys^{spec} under the precondition that $Sys^* \geq_{\mathsf{sec}} Sys^{\mathsf{spec}}$ has already been proven. In Chapter 4 we present a practical example which shows that formally verified bisimulations are well-suited for proving this relation.

We now define composition for every number n of systems $Sys_1, \ldots, Sys_n$.

Definition 2.19 *(Composition) The composition of structures and of systems is defined as follows:*

a) Structures $(\hat{M}_1, S_1), \ldots, (\hat{M}_n, S_n)$ *are* composable *if* $\mathsf{ports}(\hat{M}_i) \cap \mathsf{forb}(\hat{M}_j, S_j) = \emptyset$ *and* $S_i \cap \mathsf{free}([\hat{M}_j]) = S_j \cap \mathsf{free}([\hat{M}_i])$ *for all* $i \neq j$. *Their* composition *is then* $(\hat{M}_1, S_1)||\ldots||(\hat{M}_n, S_n) := (\hat{M}, S)$ *with* $\hat{M} = \hat{M}_1 \cup \ldots \cup \hat{M}_n$ *and* $S = (S_1 \cup \ldots \cup S_n) \cap \mathsf{free}([\hat{M}])$.

b) A system Sys *is a* composition *of* $Sys_1, \ldots, Sys_n$, *written* $Sys \in Sys_1 \times \cdots \times Sys_n$, *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* $i = 1, \ldots, n$.

c) We then call $(\hat{M}_i, S_i)$ *the* restriction *of* $(\hat{M}, S)$ *to* Sys_i *and write* $(\hat{M}_i, S_i) = (\hat{M}, S)\lceil_{Sys_i}$.

◇

The first condition for composability makes one structure a valid user of another. The second one excludes ambiguities for S, specifically the case where $p \in \mathsf{free}([\hat{M}_i]) \cap \mathsf{free}([\hat{M}_j])$ (e.g., a clock port for a high-level connection between these structures) and $p \in S_i$ but $p \notin S_j$.

We are now ready to state the composition theorem. Throughout the theorem the notation of Figure 2.3 is used.

Theorem 2.1 (Secure Two-system Composition) *Let Sys_0, Sys'_0, Sys_1 be systems and $Sys_0 \geq^{f_0} Sys'_0$ for a valid mapping f_0.*

Let $Sys^{\#} \in Sys_0 \times Sys_1$ and $Sys^* \in Sys'_0 \times Sys_1$ be compositions that fulfill the following structural conditions: For every structure $(\hat{M}^{\#}, S) \in Sys^{\#}$ with restrictions $(\hat{M}_i, S_i) = (\hat{M}^{\#}, S)\lceil_{Sys_i}$ 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, lies in Sys^*, and fulfills $\mathsf{ports}(\hat{M}'_0) \cap S_1^c = \mathsf{ports}(\hat{M}_0) \cap S_1^c$. Let $f^{\#}$ denote the function that maps each $(\hat{M}^{\#}, S)$ to the set of these compositions. Then we have

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

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

The proof can be found in [49]. However, we briefly sketch the proof technique because we will perform similar proofs throughout the thesis. The proof can be illustrated by Figure 2.4.

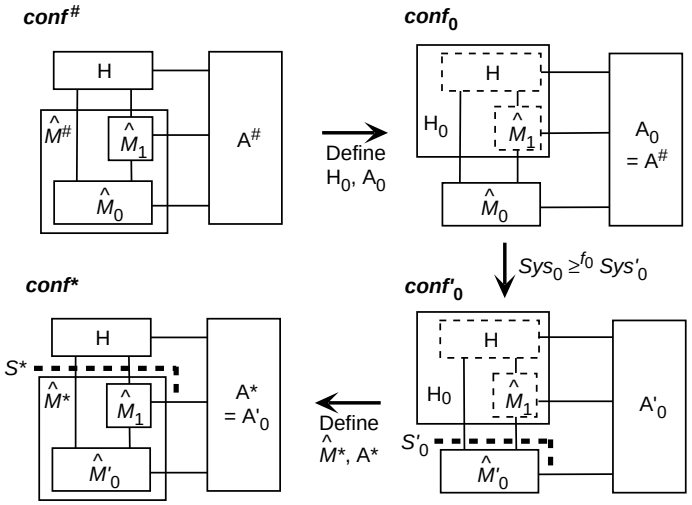

Figure 2.4: Configurations in the composition theorem. Dashed machines are internal submachines. (The connections drawn inside H_0 are not dashed because the combination does not hide them.)

In the beginning an arbitrary configuration $conf^{\#} \in \mathsf{Conf}^{f^{\#}}(Sys^{\#})$ is given. In the first step, the machines belonging to Sys_1 are combined with the honest user H yielding a new honest user H_0. After defining a new adversary A_0 (which equals the old one $\mathsf{A}^{\#}$ in this case) this gives a new configuration $conf_0 \in \mathsf{Conf}(Sys_0)$, and we

show that this new configuration yields identical views for the original honest user H as the original configuration $conf^{\#}$. After showing that $conf_0$ is suitable, the precondition $Sys_0 \geq_{\mathsf{sec}} Sys_0'$ can be applied yielding an indistinguishable configuration $conf_0' \in \mathsf{Conf}(Sys_0)$. The user H_0 is now split into its original machines again, finally yielding the desired indistinguishable configuration $conf^* \in \mathsf{Conf}^{f^*}(Sys^*)$ for a newly defined adversary A^* (which equals A_0'). The claim now follows from the transitivity of indistinguishability of views.

Combining machines and the honest user to a new user is an essential proof technique that will be used again later on, so the reader is well advised to become familiar with that. Finally, we have the following corollary.

Corollary 2.1 *Consider five systems satisfying the preconditions of Theorem 2.1, and a sixth one, Sys^{spec}, with $Sys^* \geq^{f_1} Sys^{\mathsf{spec}}$. Then $Sys^{\#} \geq^{f} Sys^{\mathsf{spec}}$ where $f := f_1 \circ f^{\#}$ as in the transitivity lemma.* □

2.3 The System for Secure Message Transmission

In this section, we review both the ideal and real system for secure message transmission in asynchronous networks [49]. It serves as a building block for many upcoming examples of this work.

2.3.1 The Ideal System

We start with an informal description of the ideal system for secure message transmission. The system is of the typical form $Sys_{\mathsf{id}} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) | \mathcal{H} \in \mathcal{ACC}\}$ and $\mathcal{ACC}$ is the powerset of $\{1, \ldots, n\}$. The system is illustrated in Figure 2.5. The ideal machine $\mathsf{TH}_{\mathcal{H}}$ models initialization and sending and receiving of messages. A user u can initialize communications with other users by inputting a command of the form $(\mathsf{snd\_init})$ to the port $\mathsf{in}_u?$ of $\mathsf{TH}_{\mathcal{H}}$. In real systems initialization corresponds to key generation and authenticated key exchange. Sending of a message to a user v is triggered by a command (send, m, v). If v is honest, the message is stored in an internal array of $\mathsf{TH}_{\mathcal{H}}$ and a command $(\mathsf{send\_blindly}, i, l, v)$ is output to the adversary, where l and i denote the length of the message m and its position in the array, respectively. This models that the adversary will see that a message has been sent and he might also be able to know the length of that message. We speak of tolerable imperfections that are explicitly granted to the adversary. Because of the underlying asynchronous timing model, $\mathsf{TH}_{\mathcal{H}}$ has to wait for a special term $(\mathsf{receive\_blindly}, v, i)$ or $(\mathsf{rec\_init}, u)$ sent by the adversary signaling that the i-th stored message sent by u to v should be delivered or that a connection between u and v should be initialized, respectively. The user v will receive outputs of the form $(\mathsf{receive}, u, m)$ and $(\mathsf{rec\_init}, u)$, respectively. If v is dishonest, $\mathsf{TH}_{\mathcal{H}}$ will simply output (send, m, v) to the adversary. Finally, the adversary can send a message m to a user u by sending a command $(\mathsf{receive}, v, m)$ to the port $\mathsf{from\_adv}_u?$ of $\mathsf{TH}_{\mathcal{H}}$ for a corrupted user v, and he can also stop the machine of any user by sending a command (stop) to a corresponding port of $\mathsf{TH}_{\mathcal{H}}$ which corresponds to exceeding the machine's runtime bounds in the real world.

Scheme 2.1 (Ideal System for Secure Message Transmission) Let $n \in \mathbb{N}$, a finite index set Σ, and a polynomial $L \in \mathbb{N}[x]$ be given. $L(k)$ bounds the length of the messages (denoted as $\mathsf{len}(m)$) for the security parameter k. Let $\mathcal{M} := \{1, \ldots, n\}$

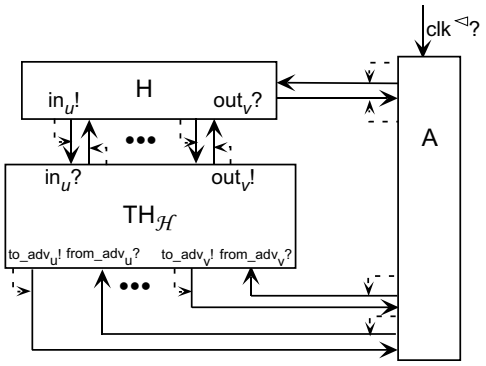

Figure 2.5: A Configuration of the Ideal System Sys_{id} for Secure Message Transmission. Normal arrows denote simple ports, dashed arrows denote clock-out ports.

denote the set of possible participants, and let the access structure $\mathcal{ACC}$ be the powerset of $\mathcal{M}$. Our specification for secure message transmission is now a localized ideal system

$$Sys_{\mathsf{id}} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \in \mathcal{ACC}\},$$

with $S^c_{\mathcal{H}} = \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$. The machine $\mathsf{TH}_{\mathcal{H}}$ is defined as follows. When $\mathcal{H}$ is clear from the context, let $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$ denote the indices of dishonest users. The ports of $\mathsf{TH}_{\mathcal{H}}$ are $\{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{\mathsf{from\_adv}_u?, \mathsf{to\_adv}_u!, \mathsf{to\_adv}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$ (cf. Figure 2.5).

$\mathsf{TH}_{\mathcal{H}}$ maintains arrays $(init^*_{u,v})_{u,v\in\mathcal{M}}$ and $(stopped^*_u)_{u\in\mathcal{H}}$ over $\{0,1\}$, both initialized with 0 everywhere, and an array $(deliver^*_{u,v})_{u,v\in\mathcal{H}}$ of lists, all initially empty. The state-transition function of $\mathsf{TH}_{\mathcal{H}}$ is defined by the following rules, written in a pseudo-code language. For the sake of readability, we exemplarily annotate the first transition of the definition, the "Send initialization" transition, i.e., key generation in the real world.

Initialization.

- **Send initialization:**

  Assume that the user u wants to generate its encryption and signature keys and distribute the corresponding public keys over authenticated channels. He can do so by sending a command ($\mathsf{snd\_init}$) to $\mathsf{TH}_{\mathcal{H}}$. Now, the system checks that the user's machine itself has not reached its runtime bound (i.e., it has not been stopped), and that no key generation of this user has already occurred in the past. These two checks correspond to $stopped^*_u = 0$ and $init^*_{u,u} = 0$, respectively. If both checks hold, the keys are distributed over authenticated channels, modeled by an output ($\mathsf{snd\_init}$) to the adversary. After receiving this command, the adversary can decide whether it schedules the keys immediately, later on, or even leave them on the channels forever. In our pseudo-code language this is expressed as follows:

  On input ($\mathsf{snd\_init}$) at $\mathsf{in}_u?$: If $stopped^*_u = 0$ and $init^*_{u,u} = 0$, set $init^*_{u,u} := 1$, and output ($\mathsf{snd\_init}$) at $\mathsf{to\_adv}_u!$, and 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$.

  The following parts should now be understood similarly:

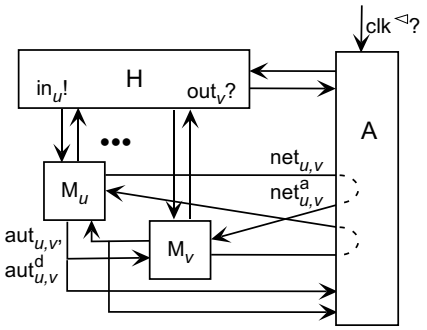

Figure 2.6: A Configuration of the Real System Sys_{real} for Secure Message Transmission.

- **Receive initialization.** On input $(\mathsf{rec\_init}, u)$ at $\mathsf{from\_adv}_v?$ with $u \in \mathcal{M}, v \in \mathcal{H}$: If $stopped^*_v = 0$ and $init^*_{u,v} = 0$ and $[u \in \mathcal{H} \Rightarrow init^*_{u,u} = 1]$, set $init^*_{u,v} := 1$ and output $(\mathsf{rec\_init}, u)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

Sending and receiving messages.

- **Send.** On input (send, m, v) at $\mathsf{in}_u?$ with $m \in \Sigma^+$, $l := \mathsf{len}(m) \leq L(k)$, and $v \in \mathcal{M} \setminus \{u\}$: If $stopped^*_u = 0$, $init^*_{u,u} = 1$, and $init^*_{v,u} = 1$:
  If $v \in \mathcal{A}$ then { output (send, m, v) at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$ } else $\{i := \mathsf{size}(deliver^*_{u,v}) + 1$;[3] $deliver^*_{u,v}[i] := m$, and output $(\mathsf{send\_blindly}, i, l, v)$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$. $\}$.

- **Receive from honest party** u**.** On input $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}_v?$ with $u, v \in \mathcal{H}$: If $stopped^*_v = 0$, $init^*_{v,v} = 1$, $init^*_{u,v} = 1$, and $m := deliver^*_{u,v}[i] \neq \downarrow$, then output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

- **Receive from dishonest party** u**.** On input $(\mathsf{receive}, u, m)$ at $\mathsf{from\_adv}_v?$ with $u \in \mathcal{A}$, $m \in \Sigma^+$, $\mathsf{len}(m) \leq L(k)$, and $v \in \mathcal{H}$: If $stopped^*_v = 0$, $init^*_{v,v} = 1$ and $init^*_{u,v} = 1$, then output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

- **Stop.** On input (stop) at $\mathsf{from\_adv}'_u?$ with $u \in \mathcal{H}$, set $stopped^*_u = 1$ and output (stop) at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

$\diamond$

2.3.2 The Real System

After presenting the abstract specification, we now briefly sketch a concrete implementation for secure message transmission. For understanding it is sufficient to give a brief review of Sys_{real}. The system is a standard cryptographic system of the form

$$Sys_{\mathsf{real}} = \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \in \mathcal{ACC}\}.$$

$\mathcal{ACC}$ is the powerset of $\mathcal{M}$, i.e., any subset of participants may be dishonest. It uses asymmetric encryption and digital signatures as cryptographic primitives. A user u can let his machine create signature and encryption keys that are sent to other users

[3] The function size denotes the size of the considered list, i.e., the number of contained elements.

over authenticated channels $\mathsf{aut}_{u,v}$. Furthermore, messages sent from user u to user v will be signed and encrypted by M_u and sent to M_v over an insecure channel $\mathsf{net}_{u,v}$, representing the net in the real world (cf. Figure 2.6). The adversary is able to schedule the communication between the users, and he can furthermore send arbitrary messages m to arbitrary users u for a dishonest sender v. In [49], it has already been shown that

$$Sys_{\mathsf{real}} \geq^{f,\mathsf{poly}}_{\mathsf{sec}} Sys_{\mathsf{id}}$$

holds for the canonical mapping f, i.e., the concrete implementation is computationally at least as secure as the specification.

Chapter 3

Some Variants of the Model

The content of this chapter can be split into two parts. In the first part, we introduce three variants of simulatability, and prove them to be equivalent to our standard definition. If we consider these variants on their own they do not really seem to be significant by now, but they greatly facilitate further proofs since we can always use the definition which is suited best for the considered problem. The second part is dedicated to the relationship between our underlying asynchronous model and its synchronous predecessor [47]. In practice, lots of protocols are synchronous, i.e., they proceed in rounds. Thus, the synchronous model is still essential to cope with these protocols. Hence, we want to drive double tracked, but without proving each and every theorem for both models. A possibility to circumvent this problem is to show that the synchronous model can be regarded as a special case of the asynchronous model, which we do not have to consider separately. This chapter contains the first, essential step of this task: we show that synchronous systems can be embedded into asynchronous ones such that simulatability is preserved by this embedding. This allows as to carry over most of the lemmas from the asynchronous to the synchronous case without proving them twice. As an example, we will show that the asynchronous version of the transitivity lemma implies in the synchronous version. However, in order to carry over lemmas dealing with composition of systems, additional work is still needed which we will only consider as future research here since we would digress too far from our actual goal of sound protocol verification.

Finally, we briefly state that this chapter contains several tedious and technical proofs, so if the reader is not really familiar with the underlying model or mainly interested in the overall results, we advise him (at least for the first reading) to move on to the next chapter and use this chapter as a reference only.

3.1 One A-H Connection

In this section we show that without loss of generality we can restrict our attention to configurations with only one "self-scheduled duplex" connection between the honest user and the adversary. This means that each of the two machines can communicate over exactly one input and one self-scheduled output port connected to the other machine. We will speak of *one*-A-H-*configurations* in this case. Moreover, the restriction to these configurations yields a new variant of simulatability, which we call *one*-A-H-*simulatability*, denoted by $\geq_{A\text{-}H}$. We will show that every configuration which does

not fulfill this precondition can be modified by replacing both the honest user and the adversary so that the following holds.

- There is only one self-scheduled duplex connection between the new user and the new adversary.
- Both the honest user and the adversary use the original user and adversary as a blackbox submachine, respectively, so that the views of the original machines are identical in both configurations.

Furthermore, we will also be able to reverse our construction after applying simulatability to restore an indistinguishable configuration for the original honest user. This serves as the main part for proving our newly defined simulatability variant to be equivalent to the standard definition. Thus, for proving simulatability between two systems, it is sufficient to restrict our attention to one-A-H-configurations.

3.1.1 Definitions

We now formally state how our new definitions are derived from the original ones.

Definition 3.1 *(One-A-H-Configurations) A* one-A-H-configuration *is a usual configuration* $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A})$ *where additionally the following properties hold:*

1. *We have*
$$|\{\mathsf{p}! \mid \mathsf{p}! \in \mathsf{ports}(\mathsf{H}) \wedge \mathsf{p}? \in \mathsf{ports}(\mathsf{A})\}| = 1$$
and
$$\mathsf{p}! \in \mathsf{ports}(\mathsf{H}) \wedge \mathsf{p}? \in \mathsf{ports}(\mathsf{A}) \Rightarrow \mathsf{p}^{\triangleleft}! \in \mathsf{ports}(\mathsf{H}),$$
i.e., we have exactly one output port from H *to* A *and the corresponding clockout port.*

2. *Similarly, we have*
$$|\{\mathsf{p}! \mid \mathsf{p}! \in \mathsf{ports}(\mathsf{A}) \wedge \mathsf{p}? \in \mathsf{ports}(\mathsf{H})\}| = 1$$
and
$$\mathsf{p}! \in \mathsf{ports}(\mathsf{A}) \wedge \mathsf{p}? \in \mathsf{ports}(\mathsf{H}) \Rightarrow \mathsf{p}^{\triangleleft}! \in \mathsf{ports}(\mathsf{A}),$$
so we have exactly one output port from A *to* H *and the corresponding clockout port yielding the desired duplex channel.*

We will in the following denote the ports of the duplex channel by $\mathsf{p}_{\mathsf{A\_H}}?, \mathsf{p}_{\mathsf{H\_A}}!, \mathsf{p}_{\mathsf{H\_A}}{}^{\triangleleft}! \in \mathsf{ports}(\mathsf{H})$ *and* $\mathsf{p}_{\mathsf{H\_A}}?, \mathsf{p}_{\mathsf{A\_H}}!, \mathsf{p}_{\mathsf{A\_H}}{}^{\triangleleft}! \in \mathsf{ports}(\mathsf{A})$.[1] *The set of these configurations is denoted by* $\mathsf{Conf}_{\mathsf{A\_H}}(Sys)$*, the set of polynomial-time one-A-H-configurations by* $\mathsf{Conf}_{\mathsf{A\_H},\mathsf{poly}}(Sys)$*. For a valid mapping* f*, suitable one-A-H-configurations are defined as usual.* ◇

[1] This is just a notation convention. We assume that these ports are new ports of every configuration, i.e., the ports are not used inside of the system itself. Otherwise, we can always achieve the desired situation by simply renaming the ports of the system before completing a structure to a configuration.

Remark 3.1. In the following we will often provide H and A with an additional index, here $_{\mathsf{A\_H}}$, if we consider specific configurations, i.e., we may write $(\hat{M}, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H}})$ instead of $(\hat{M}, S, \mathsf{H}, \mathsf{A})$. We hope that this measure improves readability because it decreases the chance of getting lost in the very long proofs. To maintain this notational convention we will often have to rename certain machines, e.g., H to $\mathsf{H}_{\mathsf{A\_H}}$, even if they are left unchanged in the constructed configuration. Such a step should obviously result in identical views of H and $\mathsf{H}_{\mathsf{A\_H}}$, so in this particular context renaming always means that we imagine H as a blackbox submachine of $\mathsf{H}_{\mathsf{A\_H}}$ so that $\mathsf{H}_{\mathsf{A\_H}}$ has exactly the same ports, and that it only forwards in- and outputs to H. Alternatively, we could rename the whole 6-tuple representing the machine. This would allow us to avoid changing the machine name which would immediately result in different views (cf. Definition 2.6). ∘

We now introduce the notion of *one*-A-H-*simulatability*, denoted by $\geq_{\mathsf{A\_H}}$. It is derived from the standard one by restricting the set of considered configurations to one-A-H-configurations.

Definition 3.2 *(One-*A-H*-Simulatability) Let two systems Sys_1 and Sys_2 with a valid mapping f be given.*

*a) We say $Sys_1 \geq^{f,\mathsf{perf}}_{\mathsf{A\_H},\mathsf{sec}} Sys_2$ (*perfectly at least as $\mathsf{A\_H}$-secure as*) if for every configuration $conf_{\mathsf{A\_H},1} = (\hat{M}_1, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H},1}) \in \mathsf{Conf}^f_{\mathsf{A\_H}}(Sys_1)$, there exists a configuration $conf_{\mathsf{A\_H},2} = (\hat{M}_2, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H},2}) \in \mathsf{Conf}_{\mathsf{A\_H}}(Sys_2)$ with $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ such that*

$$view_{conf_{\mathsf{A\_H},1}}(\mathsf{H}_{\mathsf{A\_H}}) = view_{conf_{\mathsf{A\_H},2}}(\mathsf{H}_{\mathsf{A\_H}}).$$

*b) We say $Sys_1 \geq^{f,SMALL}_{\mathsf{A\_H},\mathsf{sec}} Sys_2$ (*statistically at least as $\mathsf{A\_H}$-secure as*) for a class $SMALL$ if the same as in a) holds with $view_{conf_{\mathsf{A\_H},1},l}(\mathsf{H}_{\mathsf{A\_H}}) \approx_{SMALL} view_{conf_{\mathsf{A\_H},2},l}(\mathsf{H}_{\mathsf{A\_H}})$ for all polynomials l, i.e., statistical indistinguishability of all families of l-step prefixes of the views.*

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

As in the standard definition, we call $conf_{\mathsf{A\_H},2}$ an indistinguishable configuration *for $conf_{\mathsf{A\_H},1}$.*

We will in the following explicitly state which definition indistinguishability refers to if it is not immediately clear from the context. Where the difference between the types of security is irrelevant, we write $\geq^f_{\mathsf{A\_H},\mathsf{sec}}$ as usual, and we omit the indices f and sec if they are clear from the context.

◇

3.1.2 Proof of Equivalence

Before we turn our attention to the actual proof of equivalence, we state the following two essential lemmas.

Lemma 3.1 *Let an arbitrary system Sys and an arbitrary configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(Sys)$ be given. Then, we can define a new honest user $\mathsf{H}_{\mathsf{A\_H}}$ and a new adversary $\mathsf{A}_{\mathsf{A\_H}}$ that use the original machines H and A as black-box submachines, respectively, such that the following holds:*

1. *The configuration $conf_{\mathsf{A\_H}} := (\hat{M}, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H}})$ is a one-A-H-configuration, i.e., $conf_{\mathsf{A\_H}} \in \mathsf{Conf}_{\mathsf{A\_H}}(Sys)$.*
2. *The view of H is identical in both configurations, i.e., we have*
$$view_{conf}(\mathsf{H}) = view_{conf_{\mathsf{A\_H}}}(\mathsf{H}).$$
3. *$conf_{\mathsf{A\_H}}$ is polynomial-time iff $conf$ is polynomial-time.*

□

Proof (sketch). Since the full proof is technical, and quite tedious, we postpone it to Appendix A.1. Instead, we only give a brief sketch here. Given the configuration $conf$ we first define a new user $\mathsf{H}_{\mathsf{A\_H}}$ according to Figure 3.1. Informally speaking, every simple port of H which has been connected to A (i.e., input ports *and* output ports) is replaced by a self-loop channel. Moreover, $\mathsf{H}_{\mathsf{A\_H}}$ has additional ports for the desired duplex channel. The main idea is that the buffers of these self-loops correspond to the original buffers between H and A. If one of these buffers $\tilde{\mathsf{p}}$ is scheduled (it does not depend by which machine), the user $\mathsf{H}_{\mathsf{A\_H}}$ delivers the message to its recipient.[2] More precisely, it simply outputs the message if it is intended for a machine of the considered structure. If the recipient is the adversary, it encodes the message m and the port name $\mathsf{p}?$ and sends it over the duplex channel. The new adversary $\mathsf{A}_{\mathsf{A\_H}}$ will decompose the message and use its blackbox-submachine A with input m at $\mathsf{p}?$. The main problem is that the original adversary A might output nonempty values at multiple ports connected to H in one transition, i.e., multiple buffers have to be "filled". In $conf_{\mathsf{A\_H}}$, this is modeled iteratively, i.e., the adversary $\mathsf{A}_{\mathsf{A\_H}}$ outputs the first such message to $\mathsf{H}_{\mathsf{A\_H}}$ which outputs it to the corresponding buffer. After that, $\mathsf{H}_{\mathsf{A\_H}}$ explicitly gives back control to $\mathsf{A}_{\mathsf{A\_H}}$ which can now send the second message and so on, until all message are finally written into their corresponding buffer. Obviously, neither the original user H nor the original adversary A can notice this iteration since they are not switched during the iteration, which finally yields identical views. ■

Lemma 3.2 *Let an arbitrary system Sys and an arbitrary configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(Sys)$ be given. Moreover, let $conf_{\mathsf{A\_H}} = (\hat{M}, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H}}) \in \mathsf{Conf}_{\mathsf{A\_H}}(Sys)$ denote the configuration which we obtain if we apply Lemma 3.1 to $conf$. Then for every configuration $conf_{\mathsf{A\_H},*}$ of the form $conf_{\mathsf{A\_H},*} = (\hat{M}, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_1)$ (i.e., the configuration $conf_{\mathsf{A\_H}}$, but with an arbitrary adversary A_1), there exists a configuration $conf_* = (\hat{M}, S, \mathsf{H}, \mathsf{A}_2)$ with*
$$view_{conf_{\mathsf{A\_H},*}}(\mathsf{H}) = view_{conf_*}(\mathsf{H}).$$
Moreover, $conf_*$ is polynomial-time iff $conf_{\mathsf{A\_H},*}$ is polynomial-time. Informally, this means that we can 'reverse' our construction of Lemma 3.1 for arbitrary adversaries A_1, and restore an indistinguishable configuration for H. □

[2] At first glance, this looks like "standard multiplexing", and the most natural idea might be to store messages symmetrically, i.e., both H and A store their own messages. However, this will not yield the desired result, since only the honest user will remain unchanged at simulatability, so nothing can be states about the messages stored by the adversary.

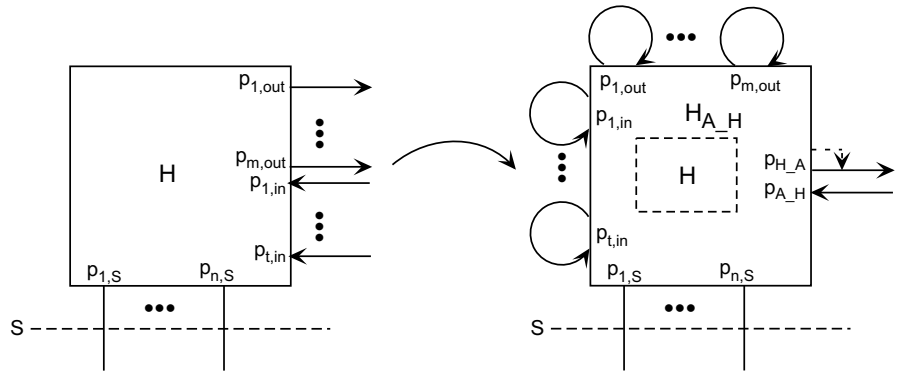

Figure 3.1: Modification of the honest user in Lemma 3.1: Ports connected to specified ports remain unchanged, simple ports connected the adversary become self-loops. Clockout ports are omitted for readability, they also remain unchanged. Finally, special ports $p_{H\_A}!$, $p_{H\_A}{}^{\triangleleft}!$ and $p_{A\_H}?$ are added.

Proof (sketch). Similar to the previous lemma, we postpone the full proof to Appendix A.1 and only sketch its idea. At first, we reverse our construction on the honest user yielding the original machine H again. The adversary A_2 has the original adversary A_1 as a blackbox submachine. The main problem is to take care of the 'iteration' between $\mathsf{H}_{\mathsf{A\_H}}$ and the adversary (cf. the proof of Lemma 3.1). Roughly speaking, $\mathsf{H}_{\mathsf{A\_H}}$ will give the control back after it has been switched by the adversary A_1, but the user H does not. Hence, in order to obtain indistinguishability, the machine A_2 models this 'giving back of control' by clocked self-loops. More precisely, $\mathsf{H}_{\mathsf{A\_H}}$ would clock the machine A_1 back again at the input port $p_{H\_A}?$, so the new adversary imitates this behaviour, i.e., it clocks its internal submachine A_2 at precisely this port $p_{H\_A}?$ using the same value. This finally yields indistinguishable view for both the original adversary A_1 and the original honest user H. ■

Theorem 3.1 *(Equivalence of standard simulatability and one-*A-H*-simulatability) Let two arbitrary systems Sys_1, Sys_2 and a valid mapping f be given. Then $Sys_1 \geq^f Sys_2$ iff $Sys_1 \geq^f_{\mathsf{A\_H}} Sys_2$. This holds for the perfect, statistical and computational case.* □

Proof. We start with the easy direction of the proof:

$\geq^f \Rightarrow \geq^f_{\mathsf{A\_H}}$: Let a suitable one-A-H-configuration $conf_{\mathsf{A\_H},1} = (\hat{M}_1, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H},1}) \in \mathsf{Conf}^f_{\mathsf{A\_H}}(Sys_1)$ be given. $conf_{\mathsf{A\_H},1}$ is a valid configuration for standard simulatability, so our precondition $Sys_1 \geq^f Sys_2$ yields a configuration $conf_{\mathsf{A\_H},2} = (\hat{M}_2, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H},2}) \in \mathsf{Conf}(Sys_2)$ with $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ and $view_{conf_{\mathsf{A\_H},1}}(\mathsf{H}_{\mathsf{A\_H}}) \approx view_{conf_{\mathsf{A\_H},2}}(\mathsf{H}_{\mathsf{A\_H}})$. We only have to show that $conf_{\mathsf{A\_H},2}$ is in fact a one-A-H-configuration, i.e., that exactly the ports of $\mathsf{A}_{\mathsf{A\_H},1}$, which were connected to $\mathsf{H}_{\mathsf{A\_H}}$ in $conf_{\mathsf{A\_H},1}$, are now ports of $\mathsf{A}_{\mathsf{A\_H},2}$ connected to $\mathsf{H}_{\mathsf{A\_H}}$. By definition, $conf_{\mathsf{A\_H},1}$ has to be a closed collection, so every port of $\mathsf{H}_{\mathsf{A\_H}}$ has to be connected either to $\mathsf{H}_{\mathsf{A\_H}}$ itself, to a specified port of the structure or to the adversary.[3]

[3]The remaining ports are unspecified ports of the system which users are forbidden to connect to. This

By definition, $\mathsf{H}_{\mathsf{A\_H}}$ remains unchanged after applying simulatability, $conf_{\mathsf{A\_H},2}$ has to be closed again, and the sets of specified ports are equal in both configurations since f is a valid mapping.

Thus, ports of the user connected to itself or to specified ports remain connected in the same way, so it may only have a connection to $\mathsf{A}_{\mathsf{A\_H},2}$ using the ports $\mathsf{p}_{\mathsf{A\_H}}?$, $\mathsf{p}_{\mathsf{H\_A}}!$, $\mathsf{p}_{\mathsf{H\_A}}{}^{\triangleleft}!$. We now finally have to show $\{\mathsf{p}_{\mathsf{H\_A}}?, \mathsf{p}_{\mathsf{A\_H}}!, \mathsf{p}_{\mathsf{A\_H}}{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{A}_{\mathsf{A\_H},2})$.

First of all, assume $\mathsf{p}_{\mathsf{H\_A}}? \in \mathsf{ports}(\mathsf{H}_{\mathsf{A\_H}})$, so the connection has to be a self-loop in $conf_{\mathsf{A\_H},2}$. However, the user has not been changed using simulatability, so the connection also was a self-loop in the first configuration which yields $\mathsf{p}_{\mathsf{H\_A}}? \in \mathsf{ports}(\mathsf{H}_{\mathsf{A\_H}})$ in $conf_{\mathsf{A\_H},1}$. This immediately yields the desired contradiction because $\mathsf{p}_{\mathsf{H\_A}}? \in \mathsf{ports}(\mathsf{A}_{\mathsf{A\_H},1})$ holds by precondition.

Similarly, $\mathsf{p}_{\mathsf{H\_A}}? \in \mathsf{ports}(\hat{M}_2)$ cannot hold because we demanded the name $\mathsf{p}_{\mathsf{H\_A}}$ to be new in every structure of the system, i.e., that the structure does not have any port to connect to it (more precisely, we have to demand that the name does not occur in any considered system, which can easily be achieved by port renaming).

Hence, $\mathsf{p}_{\mathsf{H\_A}}? \in \mathsf{ports}(\mathsf{A}_{\mathsf{A\_H},2})$ must hold. This can be proven analogously for $\mathsf{p}_{\mathsf{A\_H}}!$ and $\mathsf{p}_{\mathsf{A\_H}}{}^{\triangleleft}!$, so we obtain $\{\mathsf{p}_{\mathsf{H\_A}}?, \mathsf{p}_{\mathsf{A\_H}}!, \mathsf{p}_{\mathsf{A\_H}}{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{A}_{\mathsf{A\_H},2})$ yielding the desired duplex channel. Putting it all together, we have found an indistinguishable one-A-H-configuration of $conf_{\mathsf{A\_H},1}$ which finishes the proof of this direction.

$\geq^{f}_{\mathsf{A\_H}} \Rightarrow \geq^{f}$**:** This direction turns out to be much more complicated; however, the main work has already been anticipated by the two preceding lemmas. The proof is done in four steps illustrated in Figure 3.2.

- *Step 1:* Let a suitable configuration $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^{f}(Sys_1)$ be given. We apply Lemma 3.1 which yields a configuration $conf_{\mathsf{A\_H},1} = (\hat{M}_1, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H},1})$. Especially, we have $conf_{\mathsf{A\_H},1} \in \mathsf{Conf}_{\mathsf{A\_H}}(Sys_1)$, i.e., $conf_{\mathsf{A\_H},1}$ is a one-A-H-configuration, and the views of the honest user H are identical in both configurations, i.e.,

$$view_{conf_1}(\mathsf{H}) = view_{conf_{\mathsf{A\_H},1}}(\mathsf{H}).$$

- *Step 2:* We have to show that $conf_{\mathsf{A\_H},1}$ is a suitable one-A-H-configuration. Using Lemma 2.3, we can assume $conf_{\mathsf{A\_H},1}$ to be suitable without loss of generality, i.e., we can transform $conf_{\mathsf{A\_H},1}$ into a suitable configuration by simple port renaming which results in a one-A-H-configuration again. Moreover, Lemma 3.1 states that $conf_{\mathsf{A\_H},1}$ is polynomial-time in the computational case, since $conf_1$ is polynomial-time by precondition in this case. Now, our precondition $Sys_1 \geq^{f}_{\mathsf{A\_H}} Sys_2$ can be applied yielding a configuration $conf_{\mathsf{A\_H},2} = (\hat{M}_2, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H},2}) \in \mathsf{Conf}_{\mathsf{A\_H}}(Sys_2)$ with $(\hat{M}_2, S) \in f(\hat{M}_1, S)$, so that $view_{conf_{\mathsf{A\_H},1}}(\mathsf{H}_{\mathsf{A\_H}}) \approx view_{conf_{\mathsf{A\_H},2}}(\mathsf{H}_{\mathsf{A\_H}})$ holds. As a special case we obtain

$$view_{conf_{\mathsf{A\_H},1}}(\mathsf{H}) \approx view_{conf_{\mathsf{A\_H},2}}(\mathsf{H})$$

by part a) of Lemma 2.4.

property is guaranteed by the definition of configurations.

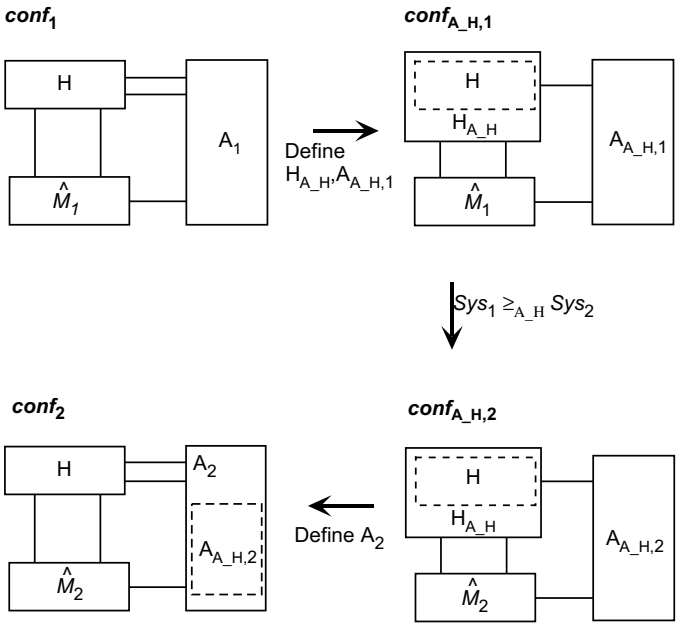

Figure 3.2: One-A-H-Simulatability implies Standard Simulatability.

- *Step 3:* In order to derive a configuration for the original honest user H, we apply Lemma 3.2 to $conf_{\mathsf{A\_H},2}$. This yields a configuration $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ such that

$$view_{conf_{\mathsf{A\_H},2}}(\mathsf{H}) = view_{conf_2}(\mathsf{H}).$$

  Moreover, $conf_2$ is polynomial-time iff $conf_{\mathsf{A\_H},2}$ is polynomial-time.

- *Step 4:* Putting it all together, we have proven

$$\begin{aligned} view_{conf_1}(\mathsf{H}) &= view_{conf_{\mathsf{A\_H},1}}(\mathsf{H}), \\ view_{conf_{\mathsf{A\_H},1}}(\mathsf{H}) &\approx view_{conf_{\mathsf{A\_H},2}}(\mathsf{H}), \\ view_{conf_{\mathsf{A\_H},2}}(\mathsf{H}) &= view_{conf_2}(\mathsf{H}). \end{aligned}$$

  Using Lemma 2.4 we can conclude $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$, so $conf_2$ is an indistinguishable configuration for $conf_1$ with respect to our standard definition of simulatability. Since $conf_1$ has been chosen arbitrary, $Sys_1 \geq^f Sys_2$ holds, which finishes our proof.

■

3.2 S-Simulatability

In this section we present another modified definition of simulatability which we again prove to be equivalent to the standard definition 2.13. Essentially, the definition only considers configurations where the adversary does not connect to any specified port.

These configurations will be called *s-configurations*, and their corresponding simulatability definition will be called *s-simulatability*. Recall that the definition of configurations explicitly excludes the case that a user connects to an unspecified port of the system; on the other hand we did not exclude the case that the adversary connects to some of the specified ports. There are good reasons for this, just imagine a person in real life that does not only act as the adversary but also as a regular user, e.g., for using services of the system in an "honest way". Such guaranteed services are usually only provided at the specified ports.

However, we will show in the following that the restriction to configurations where the honest user connects to *all* specified ports of the structure is without loss of generality. As in the previous section, we will show that every configuration which does not fulfill the preconditions of s-configurations can be modified by replacing both the honest user and the adversary so that the following holds.

- The newly defined honest user connects to all specified ports of the structure.
- The new honest user has the original user as a blackbox submachine, and the view of the original user is identical in both configurations.

Similar to the previous section, we will again be able to reverse our construction after applying s-simulatability, and to restore an indistinguishable configuration for the original honest user.

3.2.1 Definitions

We now state the new definitions.

Definition 3.3 *(S-Configurations) An* s-configuration *of a system Sys is a usual configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A})$ where additionally $S^c \subseteq \mathsf{ports}(\mathsf{H})$ must hold. The set of these configurations is denoted by $\mathsf{Conf}_{\mathsf{s}}(Sys)$. The set of polynomial-time s-configurations is denoted by $\mathsf{Conf}_{\mathsf{s,poly}}(Sys)$. For a valid mapping f, suitable s-configurations are defined as usual.* ◇

According to Remark 3.1, we will provide both the honest user and the adversary with an additional index s, writing $(\hat{M}, S, \mathsf{H}_\mathsf{s}, \mathsf{A}_\mathsf{s})$ instead of $(\hat{M}, S, \mathsf{H}, \mathsf{A})$.

Definition 3.4 *(S-Simulatability) Let two systems Sys_1 and Sys_2 with a valid mapping f be given.*

*a) We say $Sys_1 \geq^{f,\mathsf{perf}}_{\mathsf{s,sec}} Sys_2$ (*perfectly at least as s-secure as*) if for every configuration $conf_{\mathsf{s},1} = (\hat{M}_1, S, \mathsf{H}_\mathsf{s}, \mathsf{A}_{\mathsf{s},1}) \in \mathsf{Conf}^f_\mathsf{s}(Sys_1)$, there exists a configuration $conf_{\mathsf{s},2} = (\hat{M}_2, S, \mathsf{H}_\mathsf{s}, \mathsf{A}_{\mathsf{s},2}) \in \mathsf{Conf}_\mathsf{s}(Sys_2)$ with $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ such that*

$$view_{conf_{\mathsf{s},1}}(\mathsf{H}_\mathsf{s}) = view_{conf_{\mathsf{s},2}}(\mathsf{H}_\mathsf{s}).$$

*b) We say $Sys_1 \geq^{f,SMALL}_{\mathsf{s,sec}} Sys_2$ (*statistically at least as s-secure as*) for a class $SMALL$ if the same as in a) holds with $view_{conf_{\mathsf{s},1},l}(\mathsf{H}_\mathsf{s}) \approx_{SMALL} view_{conf_{\mathsf{s},2},l}(\mathsf{H}_\mathsf{s})$ for all polynomials l, i.e., statistical indistinguishability of all families of l-step prefixes of the views.*

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

$conf_2$ *is called an* indistinguishable configuration *for* $conf_1$ *with respect to s-simulatability and we will as usual omit the indices* f *and* sec *if they are clear from the context.*

◇

Before we turn our attention to the compulsory proof of equivalence we additionally define the restriction of runs to a set of ports. Although this will not be needed in the main proof, it allows us to state and prove a more general lemma which will become very useful in Chapter 5 where we consider integrity properties.

Definition 3.5 *(Restriction of Runs to Ports) Let a closed collection* $\hat{C}$ *be given. The* restriction *of a run* r *to a set* $\mathcal{S} \subseteq \mathsf{ports}(\hat{C})$ *(written* $r\lceil_{\mathcal{S}}$*) is defined by the following algorithm that modifies every step* $(name_{\mathsf{M}}, s, \mathcal{I}, s', \mathcal{O})$ *of the run as follows.*

- *If* $\mathsf{ports}(\mathsf{M}) \cap \mathcal{S} = \emptyset$*, delete the step from the run.*
- *If* $\mathcal{S}_{in} := \mathsf{in}(\mathsf{ports}(\mathsf{M})) \cap \mathcal{S} \neq \emptyset$ *and* $\mathcal{S}_{out} := \mathsf{out}(\mathsf{ports}(\mathsf{M})) \cap \mathcal{S} = \emptyset$*, replace the step of the run by* $\bigcup_{\mathsf{p}_i? \in \mathcal{S}_{in}, I_i \neq \epsilon} \{\mathsf{p}_i? : I_i\}$ *where* I_i *is the input of* M *at port* $\mathsf{p}_i?$*. In case of an empty set, i.e.,* $I_i = \epsilon$ *for all* $\mathsf{p}_i? \in \mathcal{S}_{in}$*, delete the step from the run.*
- *If* $\mathcal{S}_{in} := \mathsf{in}(\mathsf{ports}(\mathsf{M})) \cap \mathcal{S} = \emptyset$ *and* $\mathcal{S}_{out} := \mathsf{out}(\mathsf{ports}(\mathsf{M})) \cap \mathcal{S} \neq \emptyset$*, replace the step of the run by* $\bigcup_{\mathsf{p}_i! \in \mathcal{S}_{out}, O_i \neq \epsilon} \{\mathsf{p}_i! : O_i\}$ *where* O_i *is the output of* M *at port* $\mathsf{p}_i!$*. In case of an empty set, delete the step from the run.*
- *If* $\mathcal{S}_{in} := \mathsf{in}(\mathsf{ports}(\mathsf{M})) \cap \mathcal{S} \neq \emptyset$ *and* $\mathcal{S}_{out} := \mathsf{out}(\mathsf{ports}(\mathsf{M})) \cap \mathcal{S} \neq \emptyset$*, replace the step of the run by both steps of the previous two parts, i.e., by the union of* $\bigcup_{\mathsf{p}_i? \in \mathcal{S}_{in}, I_i \neq \epsilon} \{\mathsf{p}_i? : I_i\}$ *where* I_i *is the input of* M *at port* $\mathsf{p}_i?$ *and* $\bigcup_{\mathsf{p}_i! \in \mathcal{S}_{out}, O_i \neq \epsilon} \{\mathsf{p}_i! : O_i\}$ *where* O_i *is the output of* M *at port* $\mathsf{p}_i!$*. As in the above steps, empty sets are deleted.*

As in definition 2.7, we obtain a family of random variables

$$run_{\hat{C}}\lceil_{\mathcal{S}} = (run_{\hat{C},ini}\lceil_{\mathcal{S}})_{ini \in Ini_{\hat{C}}}$$

and similarly for l*-step prefixes (i.e., we first take the restriction of the run to the set* $\mathcal{S}$ *of ports, and then consider the* l*-step prefix of that restriction). If we consider a configuration* $conf$ *we write* $run_{conf,k}\lceil_{\mathcal{S}}$ *for the individual random variables according to Definition 2.9.* ◇

3.2.2 Proof of Equivalence

After introducing the new definitions we are now ready to turn our attention to the actual proof of equivalence. In order to make the proof more readable, we first present another lemma which captures the first of the four usual steps of the proof. As we already mentioned above, it will additionally play an important role in later parts of the thesis.

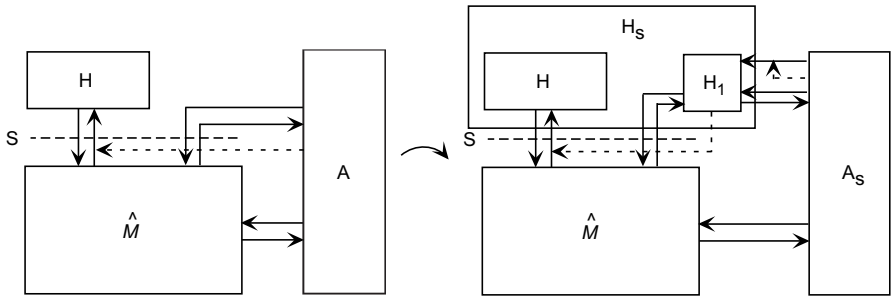

Figure 3.3: Sketch of the proof of Lemma 3.3

Lemma 3.3 *Let a system Sys be given. For every configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(Sys)$, there is a new honest user $\mathsf{H_s}$ using H as a blackbox submachine and a new adversary $\mathsf{A_s}$ such that the following holds:*

1. *$conf_\mathsf{s} := (\hat{M}, S, \mathsf{H_s}, \mathsf{A_s}) \in \mathsf{Conf_s}(Sys)$.*
2. *$view_{conf}(\mathsf{H}) = view_{conf_\mathsf{s}}(\mathsf{H})$ where the view of H in $conf_\mathsf{s}$ is given as a submachine of $\mathsf{H_s}$.*
3. *$conf_\mathsf{s}$ is polynomial-time iff $conf$ is polynomial-time.*
4. *The probability of the runs restricted to the set S of specified ports is identical in both configurations, i.e., $run_{conf}\lceil_S = run_{conf_\mathsf{s}}\lceil_S$.*

□

Proof (sketch). Since the full proof is quite technical and tedious, we postpone it to the Appendix. We only give a brief sketch how the proof is performed. We will define a new machine H_1 which is inserted between the system and the adversary, so that H_1 now exactly uses the specified ports formerly connected to A (cf. Figure 3.3). This machine will mainly forward messages, so it will not change the probability of the runs at the specified ports. Combination of H_1 and the original H will yield the intended user $\mathsf{H_s}$. The adversary $\mathsf{A_s}$ will be mainly derived by port renaming of A with the only difference that clockout ports of A have to be simulated by $\mathsf{A_s}$ in a different way, mainly by additional output ports. This will give us a configuration $conf_\mathsf{s} \in \mathsf{Conf_s}(Sys)$ as shown in the right side of Figure 3.3. However, the main difficulty of the proof (and also the reason why we postponed it) is that it has to ensure that the new honest user $\mathsf{H_s}$ is polynomial-time in case of a polynomial-time configuration. This aspect requires a thorough look at the details and significantly lengthens the size of the proof. ■

We are now ready to state and prove our main theorem. Using the results of the previous lemma, its proof turns out to be quite simple.

Theorem 3.2 *(Equivalence of standard simulatability and s-simulatability) Let two arbitrary systems Sys_1, Sys_2 and a valid mapping f be given. Then $Sys_1 \geq^f Sys_2$ iff $Sys_1 \geq_\mathsf{s}^f Sys_2$. This holds for the perfect, statistical and computational case.* □

Proof. We start with the easy direction of the proof.

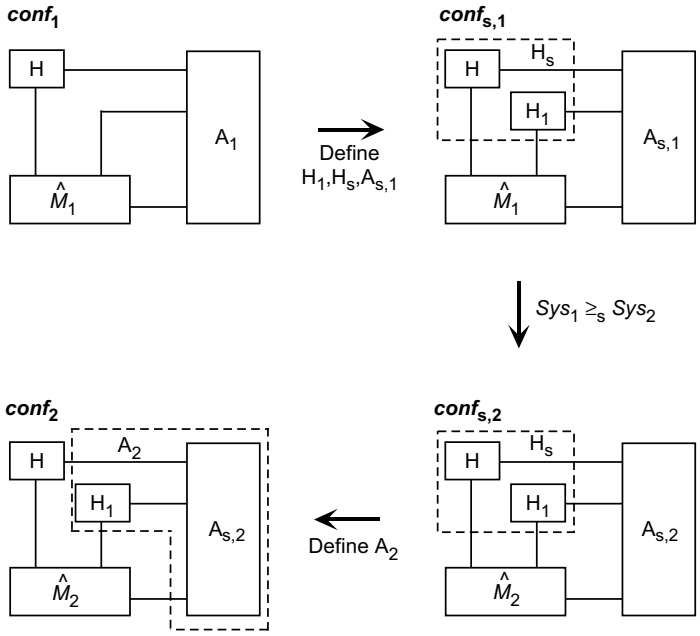

Figure 3.4: S-Simulatability implies Standard Simulatability.

$\geq^f \Rightarrow \geq_s^f$**:** Assume that $Sys_1 \geq^f Sys_2$ holds. So for every suitable configuration $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^f(Sys_1)$, there exists a configuration $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ with $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ such that $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$ holds. Since $\mathsf{Conf}_s^f(Sys_1) \subseteq \mathsf{Conf}^f(Sys_1)$ holds by definition, every suitable s-configuration $conf_{s,1} = (\hat{M}_1, S, \mathsf{H}_s, \mathsf{A}_{s,1}) \in \mathsf{Conf}_s^f(Sys_1)$ also has an indistinguishable configuration $conf_2 \in \mathsf{Conf}(Sys_2)$. Moreover, the honest user H_s remains unchanged using simulatability and $S^c \subseteq \mathsf{ports}(\mathsf{H}_s)$ holds by precondition in $conf_{s,1}$, so $S^c \subseteq \mathsf{ports}(\mathsf{H}_s)$ must also hold in $conf_2$. Thus, $conf_2 \in \mathsf{Conf}_s(Sys_2)$ which finishes this direction of the proof.

$\geq_s^f \Rightarrow \geq^f$**:** This direction will mainly be proven using Lemma 3.3. Assume that $Sys_1 \geq_s^f Sys_2$ holds and let an arbitrary suitable configuration $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^f(Sys_1)$ be given. We have to show that there is an indistinguishable configuration $conf_2 \in \mathsf{Conf}(Sys_2)$. The proof will be split in the well-known four steps, shown in Figure 3.4.

- *Step 1:* We apply Lemma 3.3 on Sys_1 immediately yielding the desired configuration $conf_{s,1} = (\hat{M}_1, S, \mathsf{H}_s, \mathsf{A}_{s,1})$. More precisely, we apply the proof of the lemma on $conf_1$ which yields the desired machines H_1 and $\mathsf{A}_{s,1}$. The lemma additionally states that the view of H does not change in both configurations, so we have

  $$view_{conf_1}(\mathsf{H}) = view_{conf_{s,1}}(\mathsf{H}).$$

  Moreover, we have $conf_{s,1} \in \mathsf{Conf}_s(Sys_1)$, and $conf_{s,1}$ is shown to be polynomial-time iff $conf_1$ is polynomial-time.

- *Step 2:* We have to show that $conf_{s,1}$ is suitable. This again immediately follows from Lemma 2.3, i.e. we can transform $conf_{s,1}$ into a suitable configuration by simple port renaming, so we can assume $conf_{s,1}$ to be suitable without loss of generality, because port renaming obviously results in a s-configuration again. Hence, our precondition $Sys_1 \geq_s^f Sys_2$ yields a s-configuration $conf_{s,2} = (\hat{M}_2, S, \mathsf{H}_s, \mathsf{A}_{s,2}) \in \mathsf{Conf}_s(Sys_2)$ with $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ such that $view_{conf_{s,1}}(\mathsf{H}_s) \approx view_{conf_{s,2}}(\mathsf{H}_s)$. As H is a submachine of H_s,

  $$view_{conf_{s,1}}(\mathsf{H}) \approx view_{conf_{s,2}}(\mathsf{H})$$

  holds by Lemma 2.4.

- *Step 3:* We now split H_s into H and H_1 again. This does not change the view of H because combination does not and indistinguishability of views is transitive.[4] Now, we combine H_1 and $\mathsf{A}_{s,2}$ into a new adversary A_2. We obtain a configuration $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ such that

  $$view_{conf_{s,2}}(\mathsf{H}) = view_{conf_2}(\mathsf{H}).$$

  In the polynomial case, A_2 has to be polynomial-time. This must hold because $\mathsf{A}_{s,2}$ and H_1 are polynomial-time by precondition and Lemma 3.3, respectively, so Lemma 2.1 applies.

- *Step 4:* Putting it all together, we have shown

  $$\begin{aligned} view_{conf_1}(\mathsf{H}) &= view_{conf_{s,1}}(\mathsf{H}), \\ view_{conf_{s,1}}(\mathsf{H}) &\approx view_{conf_{s,2}}(\mathsf{H}), \\ view_{conf_{s,2}}(\mathsf{H}) &= view_{conf_2}(\mathsf{H}). \end{aligned}$$

  Using Lemma 2.4, we can conclude $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$, so $conf_2$ is an indistinguishable configuration for $conf_1$ with respect to standard simulatability. $conf_1$ has been chosen arbitrary, so $Sys_1 \geq^f Sys_2$ holds which finishes our proof.

■

The notion of s-simulatability and Lemma 3.3 will play an important role in Chapter 5 where we define integrity properties for our reactive systems, and show that they are preserved under simulatability. This proof will make exhaustive use of Lemma 3.3.

3.2.3 Combining both Variants of Simulatability

So far, we have proven that we can either restrict ourselves on configurations with one self-scheduled duplex connection between the honest user and the adversary or on configurations where specified ports are only used by the honest user. The remaining question is whether a combination of both definitions of simulatability, i.e., restricting configurations towards both preconditions is still equivalent to the original one. We can answer this question in the affirmative, and the proof can immediately be derived from

[4] Otherwise, a combination of machines that is split again could result in a different view of internal machines as in the original configuration which surely yields a contradiction, because the configuration has not been changed.

our previous results, especially Lemma 3.3 and parts of the previous proofs. However, since we cannot directly apply our theorems and lemmas but have to "copy" parts of the proofs instead, we will only sketch the proof in the following. Missing details can be proven similarly to the previous theorems.

Proof (sketch). We denote this new definition by $\geq_{\mathsf{A\_H,s}}$. The direction $\geq^{f} \Rightarrow \geq^{f}_{\mathsf{A\_H,s}}$ is very simply and can be performed analogous to the previous theorems. For proving $\geq^{f}_{\mathsf{A\_H,s}} \Rightarrow \geq^{f}$, assume that we have an arbitrary configuration $conf_1$ of a system Sys_1.

1. At first we can transform $conf_1$ to an indistinguishable s-configuration $conf_{\mathsf{s},1} \in \mathsf{Conf_s}(Sys_1)$ applying Lemma 3.3.

2. Now, we apply the first step of the proof of Theorem 3.1 on $conf_{\mathsf{s},1}$. If we take a look at our construction of $\mathsf{H_s}$ in the actual proof we can see that the newly derived configuration $conf_{\mathsf{A\_H,s},1} \in \mathsf{Conf_{A\_H}}(Sys_1)$ does not change the connections of the honest user to the specified ports, i.e., the new honest user connects to the same subset of specified ports as the old one (i.e., it connects to *all* ports of S).

3. Thus, $conf_{\mathsf{A\_H,s},1}$ is both a s-configuration and a one-A-H-configuration, so our precondition $Sys_1 \geq^{f}_{\mathsf{A\_H,s}} Sys_2$ can be applied. The rest of the theorem can now be proved as usual, i.e., we obtain an indistinguishable configuration $conf_{\mathsf{A\_H,s},2} \in \mathsf{Conf_{A\_H,s}}(Sys_2)$.

4. The third step of the proof of Theorem 3.1 now yields an indistinguishable configuration $conf_{\mathsf{s},2} \in \mathsf{Conf}(Sys_2)$. By construction, the reversed honest user connects to the same subset of specified ports, i.e., it also connects to all specified ports. Thus, $conf_{\mathsf{s},2} \in \mathsf{Conf_s}(Sys_2)$.

5. Finally, we use the third step of the proof of Theorem 3.2 yielding an indistinguishable configuration $conf_2 \in \mathsf{Conf}(Sys_2)$ for the original honest user.

■

3.3 Guessing Outputs of the Adversary

In this section we built a relation of our model to a fundamental concept of cryptography: guessing outputs of the adversary. Guessing outputs are the main concept in many cryptographic definitions, e.g., semantic security [22] or adaptive chosen ciphertext attack [50], and play an important role in current definitions of multi-party function evaluation. Prior to this work, there already exists a synchronous version of simulatability with guessing outputs [47]. On the one hand, this version and the actual proof of equivalence are quite similar to the ones we present here, but on the other hand, the occurrence of buffers in the asynchronous model significantly complicates the notion of guessing outputs. As we proceed, we will discuss this problem in further detail.

Using the terminology of our model we consider configurations where the adversary has one emphasized output port guess!. Configurations with guessing outputs, i.e., "what an *adversary* sees is simulatable" mainly fit out intuition of privacy, whereas our standard notion "what the *user* sees is simulatable" corresponds to integrity. However, we will show that both definitions are equivalent in the reactive case, so we do not have to worry about which one to choose, i.e., which one might be more general or more

expressive. As in the last two sections we will obtain a new variant of simulatability called *guessing simulatability*, denoted by $\geq_g$.

Before we can formally define guessing configurations, we are confronted with the question what to do with the port $\mathsf{guess}!^C$. Usually, the port $\mathsf{guess}!$ is regarded as free, i.e., it connects to nowhere. This possibility does not fit our definition of configurations because the corresponding completion would no longer be closed, and hence, runs would no longer be defined. This is not an insurmountable problem since our run algorithm could as well be applied to non-closed completions as long as there are no free input ports and no clockout ports for delivering messages from buffers with free output ports.[5] We could then represent guesses of the adversary by restricting the view of A on the port $\mathsf{guess}!$.

The second possibility is to define a special machine Out to close the completion, i.e., Out simply has one port $\mathsf{guess}?$ and does nothing an arbitrary inputs at this port. It just "catches" the guessing outputs of the adversary. Now, guesses of A can either be expressed by restricting runs to the port $\mathsf{guess}!$ or to $\mathsf{guess}?$. It does not matter which one we choose because the adversary will always have the corresponding clockout port $\mathsf{guess}^{\triangleleft}!$, so any output at $\mathsf{guess}!$ can as well be scheduled by the adversary. Moreover, restricting the run to $\mathsf{guess}?$ can be used to fit our intuition of a *final* guessing output usually used in cryptographic definitions, i.e., the adversary only has one final guess which can easily be modeled by letting Out enter final state after its first input.

We decided to choose the second probability because it will not only simplify the following proof but it will also be more closely related to non-interference constructions introduced in later parts of the thesis.

The remaining question is whether Out is regarded as part of the system or as part of the user. Anticipating Chapter 6 and Chapter 7, we will now introduce so-called multi-party configurations, which generalize standard configurations to multiple users. The machine Out will then be represented as an additional second user. Our reasons not to include the machine in the system are quite simple. At first, the machine ought to be unchanged using simulatability which is a typical property of the honest user. Furthermore, it allows us to already build a bridge to future chapters of the thesis, where privacy properties will be defined using the mentioned multi-party configurations.

3.3.1 Definitions

We first introduce the definition of multi-party configurations.

Definition 3.6 *(Multi-Party Configurations) A* multi-party configuration $conf^{\mathsf{mp}}$ *of a system Sys is a tuple $(\hat{M}, S, U, \mathsf{A})$ where $(\hat{M}, S) \in Sys$ is a structure, U is a* set *of machines called users such that $\mathsf{ports}(U) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$ holds and the completion $\hat{C} := [\hat{M} \cup U \cup \{\mathsf{A}\}]$ is a closed collection.*

The set of these configurations will be denoted by $\mathsf{Conf}^{\mathsf{mp}}(Sys)$, those with a polynomial-time adversary and polynomial-time users by $\mathsf{Conf}^{\mathsf{mp}}_{\mathsf{poly}}(Sys)$ and we will omit the indices mp *and* poly *if they are clear from the context. For a valid mapping f, suitable multi-party configuration are defined as usual using the ports of U instead of the ports of the single user* H. ◇

[5] If we take a closer look at the run algorithm we can see that the first four steps can be carried through as usual. The last step could only be a problem if such a buffer with free output ports is scheduled, but this cannot happen because we demanded that the corresponding clockout port is not contained in the configuration.

Obviously, runs and views are also defined for multi-party configurations because we demanded the completion $\hat{C}$ to be closed.

Remark 3.2. Note that every multi-party configuration can be regarded as a usual configuration in a natural way simply by combining all machines of U into one user H. An immediate consequence of this embedding is that the previously proven lemmas and theorems about simulatability simply carry over to multi-party configurations since the derived indistinguishable configuration contains the same honest user which finally can be decomposed again. The rigorous proof is obviously very simply, so we simply sketch it. First of all, we combine the machines of U to a new honest user, which does not change the view of any machine of U. Now, simulatability for 'usual' configurations can be applied yielding an indistinguishable configuration of the second system for the same honest user. Finally, we split the user into its original set of users again yielding identical views which gives us the desired configuration. The only remaining step is to show that the standard configuration derived after the first step is in fact suitable which immediately follows from the precondition $\mathsf{ports}(U) \cap \mathsf{forb}(\hat{M}_i, S) = \emptyset$ for $i \in \{1, 2\}$ of suitable multi-party configurations. The opposite direction of the proof, i.e., that simulatability for multi-party configurations implies 'usual' simulatability, is trivial, because standard configurations are simply a special case with $U = \{\mathsf{H}\}$. ∘

Based on this new definition we can turn our attention to the definition of guessing configurations which are our actual topic of this section.

Definition 3.7 *(Guessing Configurations) A* guessing configuration *of a system Sys is a usual multi-party configuration $conf_{\mathsf{g}} = (\hat{M}, S, U_{\mathsf{g}}, \mathsf{A}_{\mathsf{g}})$ of the system with $U_{\mathsf{g}} := \{\mathsf{H}_{\mathsf{g}}, \mathsf{Out}\}$ so that the following two properties hold.*[6]

1. *The machine* Out *only has one input port for catching guesses of the adversary, doing nothing on arbitrary inputs at this port. In a polynomial configuration,* Out *also has to be polynomial so we assume that it enters final state after a polynomial number of steps, e.g., realized by an internal counter.*

2. *The adversary connects to the input port of* Out *and also possesses the corresponding clockout port. We will in the following denote the ports of this channel by* guess, *so*

$$\mathsf{guess?} \in \mathsf{ports}(\mathsf{Out}) \text{ and } \{\mathsf{guess!}, \mathsf{guess}^{\triangleleft}\mathsf{!}\} \subseteq \mathsf{ports}(\mathsf{A}_{\mathsf{g}})$$

   must hold.[7]

The set of all these configurations is denoted by $\mathsf{Conf}_{\mathsf{g}}(Sys)$, the set of polynomial-time configurations by $\mathsf{Conf}_{\mathsf{g,poly}}(Sys)$. Suitable guessing configurations with respect to a given mapping f carry over from Definition 3.6 as usual. For a given guessing configuration $conf_{\mathsf{g}}$ let $view_{conf_{\mathsf{g}}}(\mathsf{H}_{\mathsf{g}}, \mathsf{guess?})$ denote the two families of random variables derived by restricting the runs to the view of H_{g} and the inputs of Out *to* guess?*, i.e. the first one contains every step of H_{g} corresponding to the usual*

[6] Here, H denotes the actual user whereas Out simply catches guessing outputs from the adversary, but doing nothing by itself, i.e., it has a predefined program. According to Remark 3.1, we again provide an additional index $_{\mathsf{g}}$ for users and the adversary

[7] Fixing the name of the channel is without loss of generality because we can always achieve the desired situation by simple port renaming. It is just a notation convention to make further proofs more readable.

view, the second contains the steps of the run restricted to the port guess? *according to Definition 3.5. More formally,* $view_{conf_g}(\mathsf{H}_g, \mathsf{guess?})$ *is a family of random variables over the cross product of the probability space of the runs, i.e., we can set* $view_{conf_g}(\mathsf{H}_g, \mathsf{guess?}) := (view_{conf_g}(\mathsf{H}_g), run_{conf_g}\lceil_{\mathsf{guess?}})$. ◇

Definition 3.8 *(Simulatability with Guessing Output) Let two systems* Sys_1 *and* Sys_2 *with a valid mapping* f *be given.*

a) We say $Sys_1 \geq^{f,\mathsf{perf}}_{\mathsf{g,sec}} Sys_2$ *(*perfectly at least as g-secure as*) if for every guessing configuration* $conf_{g,1} = (\hat{M}_1, S, U_g, \mathsf{A}_{g,1}) \in \mathsf{Conf}^f_g(Sys_1)$*, there exists a configuration* $conf_{g,2} = (\hat{M}_2, S, U_g, \mathsf{A}_{g,2}) \in \mathsf{Conf}_g(Sys_2)$ *with* $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ *such that*

$$view_{conf_{g,1}}(\mathsf{H}_g, \mathsf{guess?}) = view_{conf_{g,2}}(\mathsf{H}_g, \mathsf{guess?}).$$

b) We say $Sys_1 \geq^{f,SMALL}_{\mathsf{g,sec}} Sys_2$ *(*statistically at least as g-secure as*) for a class* $SMALL$ *if the same as in a) holds with* $view_{conf_{g,1},l}(\mathsf{H}_g, \mathsf{guess?}) \approx_{SMALL} view_{conf_{g,2},l}(\mathsf{H}_g, \mathsf{guess?})$ *for all polynomials* l*, i.e., statistical indistinguishability of all families of* l*-step prefixes of the views.*

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

As in the standard definition and the prior variants, we speak of indistinguishable configurations and omit indices as usual if they are clear from the context.

◇

3.3.2 Proof of Equivalence

Theorem 3.3 *(Equivalence of standard simulatability and simulatability with guessing outputs) Let two arbitrary systems* Sys_1, Sys_2 *and a valid mapping* f *be given, then* $Sys_1 \geq^f Sys_2$ *iff* $Sys_1 \geq^f_g Sys_2$*. This holds for the perfect, the statistical and the computational case.* □

Proof. The proof will be done in the four well-known steps, so we will omit the usual proofsketch this time. Figure 3.5 shows the direction "$\geq^f \Rightarrow \geq^f_g$" we will show first. However, it may help as well to understand the opposite direction, since there are no complicated constructions of new users and adversaries in the proof, but only simple addings of new ports.

$\geq^f \Rightarrow \geq^f_g$: Let an arbitrary suitable guessing configuration $conf_{g,1} = (\hat{M}_1, S, U_g, \mathsf{A}_{g,1}) \in \mathsf{Conf}^f_g(Sys_1)$ be given. According to Figure 3.5, we define a new honest user H by combining the machines H_g and Out. The adversary remains unchanged, but in order to fit our usual notation of standard simulatability we rename it to A_1 which yields a configuration $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1)$.[8]

Combination does not change the view of H_g and Out, so we have $view_{conf_{g,1}}(\{\mathsf{H}_g, \mathsf{Out}\}) = view_{conf_1}(\{\mathsf{H}_g, \mathsf{Out}\})$. The restriction to the set

[8] Recall, that renaming means that A_1 uses $\mathsf{A}_{g,1}$ as a blackbox submachine simply forwarding in- and outputs to and from it according to Remark 3.1.

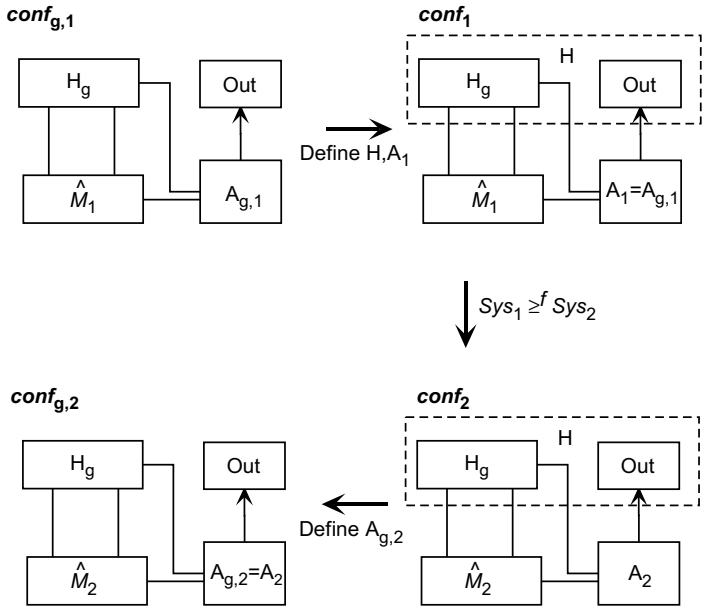

Figure 3.5: Standard simulatability implies simulatability with guessing outputs

$\{\mathsf{guess?}\}$ is a well-defined function on the view of Out by definition, so we obtain

$$view_{conf_{g,1}}(\mathsf{H_g}, \mathsf{guess?}) = view_{conf_1}(\mathsf{H_g}, \mathsf{guess?}).$$

Obviously, $conf_1$ is now a standard configuration containing one single user. Moreover, it is suitable with respect to the mapping f because $\mathsf{ports}(\mathsf{H}) = \mathsf{ports}(\{\mathsf{H_g}, \mathsf{Out}\})$ holds by construction and $\mathsf{ports}(\{\mathsf{H_g}, \mathsf{Out}\}) \cap \mathsf{forb}(\hat{M}_2, S) = \emptyset$ holds by precondition for all $(\hat{M}_2, S) \in f(\hat{M}_1, S)$ which immediately implies $\mathsf{ports}(\mathsf{H}) \cap \mathsf{forb}(\hat{M}_2, S) = \emptyset$. Thus, our precondition $Sys_1 \geq^f Sys_2$ can be applied yielding a configuration $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ such that

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

This again contains

$$view_{conf_1}(\mathsf{H_g}, \mathsf{guess?}) \approx view_{conf_2}(\mathsf{H_g}, \mathsf{guess?})$$

as a special case.

Now, we split H into $\mathsf{H_g}$ and Out again which does not change their view. The adversary $\mathsf{A}_{g,2}$ equals A_2 yielding a configuration $conf_{g,2} = (\hat{M}_2, S, U_g, \mathsf{A}_{g,2})$ with $view_{conf_2}(\{\mathsf{H_g}, \mathsf{Out}\}) = view_{conf_{g,2}}(\{\mathsf{H_g}, \mathsf{Out}\})$.

Thus,

$$view_{conf_2}(\mathsf{H_g}, \mathsf{guess?}) = view_{conf_{g,2}}(\mathsf{H_g}, \mathsf{guess?})$$

holds as a special case and we can conclude

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

combining the results of the previous steps.

We finally have to show that $conf_{g,2}$ is in fact a guessing configuration. Obviously, it fulfills the precondition with respect to the machine Out because it has not been changed during the previous steps. Moreover, $\mathsf{guess}!$ and $\mathsf{guess}^{\triangleleft}!$ are in fact ports of the configuration, because $\mathsf{guess}? \in \mathsf{ports}(\mathsf{Out})$ holds by construction and $conf_2$ has to be closed by definition. This now immediately implies $\{\mathsf{guess}!, \mathsf{guess}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{A}_2)$, because the ports of the user have not been changed after simulatability, so either H_g or Out would already have had one of these ports in the original configuration $conf_{g,1}$ yielding a contradiction. Furthermore, they cannot be ports of the system because we assume the name guess to be new among all port names of the system. Otherwise, we can give them new names that do not occur in the system before applying simulatability.

Thus, $conf_{g,2} \in \mathsf{Conf}_g(Sys_2)$ which finishes this direction of the proof.

$\geq_g^f \Rightarrow \geq^f$: Let now an arbitrary configuration $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}^f(Sys_1)$ be given. We now construct a guessing configuration $conf_{g,1} = (\hat{M}_1, S, U_g, \mathsf{A}_{g,1}) \in \mathsf{Conf}_g(Sys_1)$ with $U_g := \{\mathsf{H}_g, \mathsf{Out}\}$ by adding the missing machine Out, the user H_g equals H, i.e., it has H as a blackbox submachine and simply forwards all messages to and from δ_{H}. The adversary $\mathsf{A}_{g,1}$ behaves exactly like A_1 but it has additional ports $\mathsf{guess}!$ and $\mathsf{guess}^{\triangleleft}!$. These ports will never be used in a run of the configuration because $\mathsf{A}_{g,1}$ simply forwards messages to and from its blackbox submachine A_1 to the corresponding ports. Obviously, this yields identical view for the original honest user in both configurations, i.e.,

$$view_{conf_1}(\mathsf{H}) = view_{conf_{g,1}}(\mathsf{H}).$$

Moreover, the configuration is suitable because the original is, and the port name guess is assumed to be disjoint with all port names of the system and of the honest user, i.e., it has to connect to the adversary. Now, our precondition $Sys_1 \geq_g^f Sys_2$ can be applied yielding a configuration $conf_{g,2} = (\hat{M}_2, S, U_g, \mathsf{A}_{g,2}) \in \mathsf{Conf}_g(Sys_2)$ with $view_{conf_{g,1}}(\mathsf{H}_g, \mathsf{guess}?) \approx view_{conf_{g,2}}(\mathsf{H}_g, \mathsf{guess}?)$.

As a special, case we restrict the family of random variables to the submachine H of H_g yielding

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

Finally, we combine Out and $\mathsf{A}_{g,2}$ to a new adversary A_2 which leaves the view of the H_g unchanged. After renaming H_g into H again we obtain the final configuration $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2)$ with

$$view_{conf_{g,2}}(\mathsf{H}) = view_{conf_2}(\mathsf{H}).$$

Obviously, $conf_2 \in \mathsf{Conf}(Sys_2)$ holds and combining all steps we conclude that $conf_2$ is an indistinguishable configuration for $conf_1$ which finishes our proof.

■

3.4 Relation to Synchronous Systems

By now, we only focused on asynchronous reactive systems. However, there is also a synchronous model introduced in [47] which can be seen as a predecessor of the asynchronous one, i.e., the synchronous model is based on the same concept of machines,

systems, configurations and so on. The main difference is that there are no clocking ports and no buffers which have only been included to model asynchronous timing. Instead, runs are defined using *rounds* which is the usual concept in synchronous scenarios. Every global round is again divided into n so-called subrounds, and there is a mapping κ from the set $\{1, \ldots, n\}$ into the powerset of considered machines, i.e., the machines of the structure, the user, and the adversary. $\kappa(i)$ denotes which machines switch in subround i. After finishing the n-th subround, the run starts the first subround of the next global round. At the beginning of each subround, all messages from the previous subround are transported from the output ports to the connected input ports. After that, each machine of $\kappa(i)$ switches with its current inputs yielding a finite distribution over the set of states and the set of possible outputs as usual.

An often asked question is whether the synchronous model is a special case of the asynchronous one, i.e., whether synchronous systems can be embedded into asynchronous ones such that simulatability is preserved. Moreover, lots of protocols in practice are synchronous, so the synchronous model is still essential to cope with these protocols. Hence, we want to drive double tracked, but without proving each and every theorem for both models.

Therefore, we present such an embedding in the following, which serves as the first, essential step of regarding the synchronous model as a special case only, which we do not have to consider explicitly. More precisely, we will show that asynchronous simulatability among these asynchronous representations implies synchronous simulatability in the computational case. Moreover, the converse implication holds in the computational case under some additional reasonable assumptions. In the perfect and statistical case both claims also hold if we perform one of the following three slight modifications:

1. We bound the number of steps the adversary can perform by an arbitrary bound. This does not really seem to be a restriction if we consider protocols in real life, because no machine will be able to run forever. Note, that we do not require the adversary to be polynomial-time as in the computational case, we only demand it to be bounded by an *arbitrary* function.

2. We restrict our model to adversaries that cannot perform infinite successive clocked self-loops. We speak of a clocked self-loop if the adversary schedules a port to itself, so that the clocked buffer can in fact deliver a message. In this case the adversary will be clocked again, so it may again perform such a (now successive) self-loop. Obviously, such an adversary can be validly defined and it may perform such self-loops forever, so no other machine will ever be switched again.

3. We modify our definition of machines in the following way. We define that a machine does not have to terminate and that the transition function of a machine M is not restricted to finite distributions over $States_{\mathsf{M}} \times \mathcal{O}_{\mathsf{M}}$. Putting it all together, we define that the transition function yields a (probably infinite, e.g., discrete) probability distribution over $States_{\mathsf{M}} \times \mathcal{O}_{\mathsf{M}} \cup \{\perp\}$, where $\perp$ denotes divergation of the machine, i.e., the probability associated to $\perp$ is the probability that M diverges. If the machine diverges the runs stops.

If we take a look at "typical" systems we do not have to worry about these modifications, since the second one will most likely be fulfilled anyway. However, we explicitly have to consider at least one of these modifications in order to formally prove the desired properties of the embedding.

3.4.1 A Brief Review of the Synchronous Model

We now briefly sketch the differences between the synchronous model of [47] and our asynchronous model. At first, ports, machines, and collections are defined similar to our model, except that there are no clock ports and no buffers, i.e., corresponding ports $\mathsf{p}?$ and $\mathsf{p}!$ are directly connected. The main difference is the definition of runs. Instead of our usual run algorithm (cf. Definition 2.6), we have a clocking scheme κ indicating which machines switch in which subround.

Definition 3.9 *(Clocking Scheme, Runs and Views) Given a closed collection $\hat{C}$, a clocking scheme κ is a mapping from a set $\{1,\ldots,n\}$ to the powerset of the set of all machines of $\hat{C}$, i.e., it assigns each number a subset of the machines. Given $\hat{C}$, κ, and a tuple $ini \in Ini_{\hat{C}} := \times_{\mathsf{M}\in\hat{C}}$ of initial states, runs are defined as follows: Each global round i has n subrounds. In subround $[i.j]$ all machines $\mathsf{M} \in \kappa(j)$ switch simultaneously, i.e., each state-transition function δ_{M} is applied to M's current input yielding a new state and output (probabilistically). The output at a port $\mathsf{p}!$ is available as input at $\mathsf{p}?$ until the machine with port $\mathsf{p}?$ is clocked next. If several inputs arrive until that time, they are concatenated. This gives a family of random variables*

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

More precisely, each run is a function mapping each triple $(\mathsf{M},i,j) \in \hat{C}\times\mathbb{N}\times\{1,\ldots,n\}$ to a quadruple (s,I,s',O) of the old state, inputs, new state, and outputs of machine M in subround $[i.j]$, with a symbol ϵ if M is not clocked in this subround.

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

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

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

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

and similarly for l-round prefixes.

◇

Remark 3.3. Alternatively, we can consider runs as a sequence of seven-tuples $(\mathsf{M},i,j,s,I,s',O)$ for ascending values of i and j. More formally, we first have all tuples $(\mathsf{M},1,1,s,I,s',O)$ for $\mathsf{M}\in\kappa(1)$. The order of these tuples can be chosen arbitrary since they switch simultaneously and do not influence each other. After that, we have the steps $(\mathsf{M},1,2,s,I,s',O)$ for all $\mathsf{M}\in\kappa(2)$ and so on, until we finally have steps of the form $(\mathsf{M},1,n,s,I,s',O)$ for all $\mathsf{M}\in\kappa(n)$. We then continue with $(\mathsf{M},2,1,s,I,s',O)$ etc. Obviously, this characterization of runs is equivalent to the original one (we just expanded the function), but it is better suited for our upcoming embedding proof. ◦

The remaining definitions of the synchronous model are analogous to its asynchronous counterpart except for configurations, which are slightly different. Instead of arbitrary clocking schemes as in the definition of runs of a collection, we only consider one special clocking scheme κ for runs of a configuration. This special clocking scheme

is given by $(\hat{M} \cup \{\mathsf{H}\}, \{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{A}\})$. Clocking the adversary between the correct machines is the well-known model of "rushing adversaries". In [47], it has been shown that this clocking scheme does not restrict the possibilities of the adversary, hence we can use it without loss of generality.

Moreover, we restrict ourselves to those configurations where the honest user and the adversary are only connected over one duplex channel. This is indeed no restriction to generality in the synchronous model, because out- and inputs at several ports can simply be concatenated using a separation symbol and decomposed again, respectively. In the following, we give these two channels fixed names $\mathsf{p\_A\_H}$ and $\mathsf{p\_H\_A}$, i.e., $\mathsf{p\_A\_H}!$ sends messages from A to H and vice versa.

3.4.2 Definition on the Embedding

After this brief synopsis of the synchronous model, we can turn our attention to the actual embedding. By definition, an embedding of one set into another is an injective function that respects a given relation on these sets. In our case, we consider the sets of synchronous and asynchronous systems, respectively, and the relation to preserve is simulatability. We will in the following write $\geq_{\mathsf{sync}}$ and $\geq_{\mathsf{async}}$ for simulatability in the synchronous and the asynchronous case, respectively. We start by defining the embedding function φ_{Sys} that assigns every synchronous system Sys_{sync} an asynchronous system Sys_{async}. We will afterwards show that this function preserves simulatability in the following way:

$$\varphi_{Sys}(Sys_{\mathsf{sync},1}) \geq_{\mathsf{async}} \varphi_{Sys}(Sys_{\mathsf{sync},2}) \Rightarrow Sys_{\mathsf{sync},1} \geq_{\mathsf{sync}} Sys_{\mathsf{sync},2}.$$

Unfortunately, the converse direction does not hold in general, but we will state a weaker theorem later on which is still sufficient for our purpose. Before we turn our attention to the mapping φ_{Sys}, we will define a similar mapping φ_{M} on single synchronous machines.

Definition 3.10 (Mapping φ_{M}) φ_{M} is a mapping on single synchronous machines that assigns every machine $\mathsf{M}_{\mathsf{sync}}$ an asynchronous machine $\mathsf{M}_{\mathsf{async}} := \varphi_{\mathsf{M}}(\mathsf{M}_{\mathsf{sync}})$ by the following rules:

- The ports of $\mathsf{M}_{\mathsf{async}}$ are given by $\mathsf{ports}(\mathsf{M}_{\mathsf{async}}) = \mathsf{ports}(\mathsf{M}_{\mathsf{sync}}) \cup \{\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}?\}$.
- Internally, $\mathsf{M}_{\mathsf{async}}$ maintains arrays $(input\_store_{\mathsf{M}_{\mathsf{sync}},\mathsf{p}?})_{\mathsf{p}? \in \mathsf{in}(\mathsf{ports}(\mathsf{M}_{\mathsf{sync}}))}$ over Σ^* initialized with ϵ everywhere.
- $\mathsf{M}_{\mathsf{async}}$ has the machine $\mathsf{M}_{\mathsf{sync}}$ as a blackbox submachine, i.e., it has its transition function $\delta_{\mathsf{M}_{\mathsf{sync}}}$.
- Internally, $\mathsf{M}_{\mathsf{async}}$ has exactly the states of $\mathsf{M}_{\mathsf{sync}}$ (the names of the states). Moreover, the initial and final states of both machines are equal.

Its behaviour is defined as follows.

- On input i at $\mathsf{p}?$ with $\mathsf{p}? \neq \mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}?$: It concatenates i to the element of $input\_store_{\mathsf{M}_{\mathsf{sync}},\mathsf{p}?}$. Informally, φ_{M} encloses a synchronous machine M such that the following holds: if an arbitrary input occurs at a port $\mathsf{p}?$, it appends this input to the array $input\_store_{\mathsf{p}?}$. Note, that the synchronous machine will only switch in its corresponding subround whereas asynchronous machines switch every time they are scheduled. Therefore, we have to store all inputs until the machine M is eventually switched.

- On an arbitrary input i at $\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}?$: It applies the state transition function $\delta_{\mathsf{M}_{\mathsf{sync}}}$ on the contents of the arrays $input\_store_{\mathsf{M}_{\mathsf{sync}},\mathsf{p}?}$ yielding a tuple $(s', \mathcal{O})$.[9] $\mathsf{M}_{\mathsf{async}}$ now assigns ϵ to $input\_store_{\mathsf{M}_{\mathsf{sync}},\mathsf{p}?}$ for all $\mathsf{p}? \in \mathsf{in}(\mathsf{ports}(\mathsf{M}_{\mathsf{sync}}))$, switches to the state s' and outputs the tuple $\mathcal{O}$. Note, that these steps are indeed possible, because we demanded every state of the synchronous machine to be a state of the asynchronous machine, and $\mathsf{out}(\mathsf{ports}(\mathsf{M}_{\mathsf{async}})) = \mathsf{out}(\mathsf{ports}(\mathsf{M}_{\mathsf{sync}}))$ holds by construction. This case corresponds to the scheduling of the synchronous machine. Later on, the port $\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}?$ will be connected to an explicit master scheduler with an explicit scheduling strategy that should model the rounds of the synchronous system.

Obviously, $\mathsf{M}_{\mathsf{async}}$ is polynomial-time by construction iff $\mathsf{M}_{\mathsf{sync}}$ is polynomial-time. Moreover, we define the function φ_{M} on a set $\hat{M}$ of synchronous machines by $\varphi_{\mathsf{M}}(\hat{M}) := \bigcup_{\mathsf{M}_{\mathsf{sync}} \in \hat{M}} \varphi_{\mathsf{M}}(\mathsf{M}_{\mathsf{sync}})$. ◇

Based on this definition, we now formalize the desired mapping φ_{Sys}.

Definition 3.11 (Mapping φ_{Sys}) Let on arbitrary synchronous system Sys_{sync}, a clocking scheme κ and a structure $(\hat{M}_{\mathsf{sync}}, S_{\mathsf{sync}}) \in Sys_{\mathsf{sync}}$ be given. We can now define $\varphi_{Sys}(Sys_{\mathsf{sync}})$ by applying φ_{M} to every machine of every structure, and adding a special machine $\mathsf{X}_{\mathsf{sync},\kappa}$ to every structure which depends on the synchronous clocking scheme κ. More formally, we have $Sys_{\mathsf{sync}} = \{(\hat{M}_{\mathsf{sync}}, S_{\mathsf{sync}}) \mid \mathsf{sync} \in I\}$ for a finite index set I and

$$\varphi_{Sys}(Sys_{\mathsf{sync}}) = \{(\varphi_{\mathsf{M}}(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{X}_{\mathsf{sync},\kappa}\}, S_{\mathsf{sync}}) \mid \mathsf{sync} \in I\}.$$

The machine $\mathsf{X}_{\mathsf{sync},\kappa}$ is an explicit master scheduler that has to be added to the considered structure to model the synchronous clocking scheme κ in the asynchronous system. Its ports are given by

- $\{\mathsf{p}^{\triangleleft}! \mid \mathsf{p}! \in \mathsf{ports}(\hat{M}_{\mathsf{sync}})\}$: Ports for clocking all output ports of the given structure.
- $\{\mathsf{p}^{\triangleleft}! \mid \mathsf{p}!^{c} \in \mathsf{free}(\hat{M}_{\mathsf{sync}})\}$: Ports for clocking inputs of the systems (either made by H or A).
- $\{\mathsf{p}_{\mathsf{A\_H}}{}^{\triangleleft}!, \mathsf{p}_{\mathsf{H\_A}}{}^{\triangleleft}!\}$: Ports for clocking the connection between A and H.[10]
- $\{\mathsf{p}_{\mathsf{M}}!, \mathsf{p}_{\mathsf{M}}{}^{\triangleleft}! \mid \mathsf{M} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}, \mathsf{A}\})\}$: Ports for clocking, i.e., giving control to each machine.

Internally, it maintains a variable $local\_rnd$ over $\{1, \ldots, n\}$ and a variable $global\_rnd$ over $\mathbb{N}$ both initialized with 1. For the sake of readability, we describe the behaviour of $\mathsf{X}_{\mathsf{sync},\kappa}$ using "for"-loops. This is just a notational convention that should be understood as follows: every time $\mathsf{X}_{\mathsf{sync},\kappa}$ is scheduled, it performs the next step of the loop.

1. **Schedule Current Machines:** For all machines $\mathsf{M} \in \kappa(local\_rnd)$ output $(global\_rnd, local\_rnd)$ at $\mathsf{p}_{\mathsf{M}}!$, 1 at $\mathsf{p}_{\mathsf{M}}{}^{\triangleleft}!$. The order of the switched machines can be chosen arbitrary.

[9] Note, that these arrays naturally correspond to inputs of the state transition function because they are enumerated by the input ports of $\mathsf{M}_{\mathsf{sync}}$.

[10] Note, that $\mathsf{X}_{\mathsf{sync},\kappa}$ is defined independent from the honest user H and the adversary A, so it cannot know their ports. We therefore restricted the configuration to a fixed number and fixed names of ports between H and A (cf. Section 3.4.1)

2. **Schedule Outgoing Buffers:** For all $\mathsf{M} \in \kappa(local\_rnd)$ output 1 at every port $\mathsf{p}^{\triangleleft}!$ with $\mathsf{p}! \in \mathsf{ports}(\mathsf{M})$. Here, the order of the switched machine can only be chosen arbitrary with the restriction that output ports of the adversary are scheduled first if $\mathsf{A} \in \kappa(local\_rnd)$.[11]

3. **Switch to next Round:** Set $local\_rnd := local\_rnd + 1$, if $local\_rnd > n$, set $global\_rnd := global\_rnd + 1$ and $local\_rnd := 1$. Go to Phase (1).

◇

Putting it all together, the master scheduler simulates the clocking scheme κ by first scheduling the machines that ought to switch in the particular subround (Step 1) and afterwards scheduling all buffers that could be influenced by outputs of these machines (Step 2). Afterwards, it switches to the next subround (Step 3). Moreover, we define a mapping φ_{conf} on configurations by

$$\varphi_{conf}(\hat{M}_{\mathsf{sync}}, S_{\mathsf{sync}}, \mathsf{H}, \mathsf{A}) := (\varphi_{\mathsf{M}}(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{X}_{\mathsf{sync},\kappa}\}, S_{\mathsf{sync}}, \varphi_{\mathsf{M}}(\mathsf{H}), \varphi_{\mathsf{M}}(\mathsf{A})).$$

We will in the following simply write φ instead of φ_{Sys}, φ_{M} and φ_{conf} if its meaning is clear from the context.

3.4.3 Preliminary Work for the Embedding Theorems

We now have to prove that the function φ has in fact the desired properties with respect to simulatability. Before we turn our attention to this property we first present an informal description how the proof of the first embedding theorem will be performed (there will be two of them). As usual, it will consists of the four well-known steps, illustrated in Figure 3.6.

1. Starting with a synchronous configuration $conf_{\mathsf{sync},1} \in \mathsf{Conf}(Sys_{\mathsf{sync},1})$, we apply our embedding function φ_{conf} which yields an asynchronous configuration $conf_{\mathsf{async},1} \in \mathsf{Conf}(\varphi_{Sys}(Sys_{\mathsf{sync},1}))$. We now have to built a relation between the runs in both configurations. This will be done by defining a mapping ϕ on the runs of the asynchronous system yielding runs of the synchronous system, and we will show in Theorem 3.4 that

   $$view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$$

   holds.

2. We can now apply our precondition $\varphi_{Sys}(Sys_{\mathsf{sync},1}) \geq^{f}_{\mathsf{async}} \varphi_{Sys}(Sys_{\mathsf{sync},2})$ yielding an indistinguishable configuration $conf_{\mathsf{async},2} \in \mathsf{Conf}(\varphi_{Sys}(Sys_{\mathsf{sync},2}))$, i.e., we obtain

   $$view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})) \approx view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}})),$$

   and, using Lemma 2.4,

   $$\phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) \approx \phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$

[11] This restriction will be essential in Step 3 of the proof of the embedding theorem. We ensure that the behaviour of the adversary $\mathsf{A}_{\mathsf{async},2}$ at its switching time does not depend on outputs of machines scheduled in the same subround. We will see that this problem does not occur for arbitrary other machines by construction.

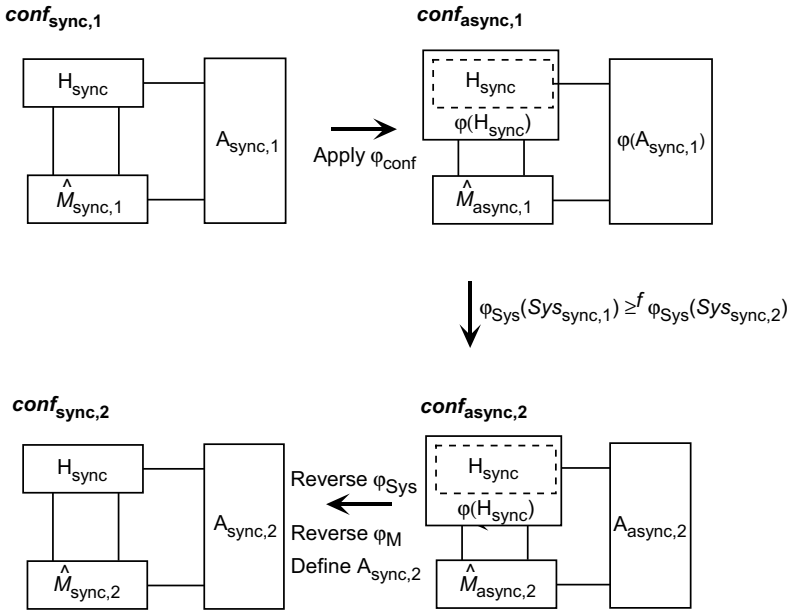

Figure 3.6: Synchronous Simulatability derived by Asynchronous Simulatability.

3. We finally want to reverse our function φ in order to obtain a synchronous system again. Obviously, both the honest user $\varphi(\mathsf{H}_{\mathsf{sync}})$ and all machines $\varphi(\hat{M}_2)$ of the structure can easily be converted back to $\mathsf{H}_{\mathsf{sync}}$ and $\hat{M}_2$, respectively, since they fit the special form of φ. In contrast to that, we do not know anything about the newly derived adversary $\mathsf{A}_{\mathsf{async},2}$, i.e., it is not forced to fit such a prescribed structure too. Therefore, we define a new adversary $\mathsf{A}_{\mathsf{sync},2}$ using $\mathsf{A}_{\mathsf{async},2}$ as a black-box submachine, and we will show in Theorem 3.5 that

$$\phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) = view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})$$

holds.

4. Altogether, this yields

$$view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}})) \approx view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}}),$$

so the claim follows.

We first take a look at the runs in a synchronous system Sys_{sync} and in its asynchronous counterpart $Sys_{\mathsf{async}} := \varphi(Sys_{\mathsf{sync}})$. We will afterwards define a mapping ϕ on runs such that

$$view_{conf_{\mathsf{sync}}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$$

holds for every synchronous configuration $conf_{\mathsf{sync}}$ and the asynchronous configuration $conf_{\mathsf{async}} := \varphi(conf_{\mathsf{sync}})$ (cf. Step 1 of our above proofsketch). In the following, we will simply write S instead of S_{sync}, because the set of specified ports is not influenced by the mapping φ.

Before we turn our attention to the actual definition of the mapping ϕ, we take a closer look at the runs of these asynchronous configurations. Thus, let an arbitrary synchronous system Sys_{sync} with a clocking scheme κ and an arbitrary configuration $conf_{\mathsf{sync}} = (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}}) \in \mathsf{Conf}(Sys_{\mathsf{sync}})$ be given. Moreover,

let an asynchronous configuration $conf_{\mathsf{async}}$ be given which fits the form $conf_{\mathsf{async}} = (\varphi(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{X}_{\mathsf{sync},\kappa}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \mathsf{A}')$ (i.e., $\varphi(conf_{\mathsf{sync}})$ but with an arbitrary adversary).

First of all, note that runs of $conf_{\mathsf{async}}$ always have a prescribed structure induced by the behaviour of the master scheduler $\mathsf{X}_{\mathsf{sync},\kappa}$, they are built by "blocks". The steps $(\mathsf{M}_{\mathsf{sync}}, i, j, s, \mathcal{I}, s', \mathcal{O})$ of the machines $\mathsf{M}_{\mathsf{sync}} \in \hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}\}$ switched in round $[i.j]$ in the synchronous run are represented by the following two blocks in the asynchronous run.

1. The first block consists of the steps induced by clocking the machines $\varphi(\mathsf{M}_{\mathsf{sync}})$ with $\mathsf{M}_{\mathsf{sync}} \in \kappa(j)$, i.e., Step (1) in the definition of $\mathsf{X}_{\mathsf{sync},\kappa}$. More precisely, the block is built by $|\kappa(j)|$ sub-blocks, one for every switched machine. Every sub-block is built by the following steps.

   - The first step of the sub-block is always given by
   $$(\mathsf{X}_{\mathsf{sync},\kappa}, s_1, \mathcal{I}_{\mathsf{c|k}^{\triangleleft}?=1}, s_1', \mathcal{O}_{\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}!=(i,j),\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}{}^{\triangleleft}!=1})$$
   for two arbitrary states s_1, s_1' of $\mathsf{X}_{\mathsf{sync},\kappa}$, i.e., the master scheduler schedules the machine $\mathsf{M}_{\mathsf{sync}}$.
   - After that, we have the two transitions of the scheduled buffer.
   - We have to distinguish the following two cases:
     – If $\mathsf{M}_{\mathsf{sync}} \neq \mathsf{A}'$ holds, there is a step
     $$(\varphi(\mathsf{M}_{\mathsf{sync}}), s, \mathcal{I}_{\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}?=(i,j)}, s', \delta_{\mathsf{M}_{\mathsf{sync}}}(input\_store_{\mathsf{M}_{\mathsf{sync}}}))$$
     and steps for the receiving buffers.
     – If $\mathsf{M}_{\mathsf{sync}} = \mathsf{A}'$ holds, we have a step
     $$(\mathsf{A}', s, \mathcal{I}_{\mathsf{p}_{\mathsf{A}}?=(i,j)}, s', \mathcal{O}).$$
     If $\mathcal{O} \neq \mathcal{O}_\epsilon$ we have steps for the receiving buffers. If there are nonempty outputs at ports $\mathsf{p}!$ and $\mathsf{p}^{\triangleleft}!$ (which has to be a self-loop because there are no free clockin ports in the system), there is furthermore a clocking step for this particular buffer. In this case, the adversary is scheduled again, so this sub-point of the block is repeated until the self-loop of the adversary either ends or it is repeated forever in case of divergation, i.e., we obtain a step $(\mathsf{A}', s', \mathcal{I}', s'', \mathcal{O})$ where $\mathcal{I}'$ is now given by $\mathcal{I}' := \mathcal{I}_{\mathsf{p}?=\mathcal{O}_{\mathsf{p}!}}$ and so on.

2. The second block consists of the steps induced by clocking the outgoing messages of the switched machines, i.e., Step (2) in the definition of $\mathsf{X}_{\mathsf{sync},\kappa}$. Now the buffers of the output ports are switched by the master scheduler. This is done similar as in the first part with the restriction that output ports of A' are clocked first if $\mathsf{A}_{\mathsf{sync}} \in \kappa(j)$. The block again has $|\kappa(j)|$ sub-blocks built by the following steps.

   - The first step of the sub-block is given by
   $$(\mathsf{X}_{\mathsf{sync},\kappa}, s_1, \mathcal{I}_{\mathsf{c|k}^{\triangleleft}?=1}, s_1', \mathcal{O}_{\mathsf{p}^{\triangleleft}!=1})$$
   for the first output port $\mathsf{p}! \in \mathsf{ports}(\mathsf{M}_{\mathsf{sync}})$ and two arbitrary states s_1, s_1' of $\mathsf{X}_{\mathsf{sync},\kappa}$.

- The step of the clocked buffer.
- In case of a nonempty output let M' denote the unique machine with $\mathsf{p}? \in \mathsf{ports}(\mathsf{M}')$. We now have to distinguish between two cases:
  - If $\mathsf{M}' \neq \mathsf{A}'$ holds, there is a step
    $$(\mathsf{M}', s, \mathcal{I}', s', \mathcal{O}_\epsilon)$$
    where $\mathcal{I}'$ consists of the output of $\varphi(\mathsf{M}_{\mathsf{sync}})$ at $\mathsf{p}!$.
  - If $\mathsf{M}' = \mathsf{A}'$ holds, we obtain a step
    $$(\mathsf{A}', s, \mathcal{I}', s', \mathcal{O})$$
    where $\mathcal{I}'$ consists of the output of $\varphi(\mathsf{M}_{\mathsf{sync}})$ at $\mathsf{p}!$. If $\mathcal{O} \neq \mathcal{O}_\epsilon$ we have steps for the receiving buffers. If $\mathcal{O}$ has a clocked self-loop (cf. the corresponding part of the first block) there is furthermore a clocking step for this particular buffer. In this case the adversary is scheduled again, so this sub-point of the block is repeated until the self-loop of the adversary either ends or it is repeated forever in case of divergation, i.e., we obtain a step $(\mathsf{A}', s', \mathcal{I}', s'', \mathcal{O})$ where $\mathcal{I}'$ is now given by $\mathcal{I}' := \mathcal{I}_{\mathsf{p}?=\mathcal{O}_{\mathsf{p}!}}$, we again obtain steps for the buffers and so on.
- The three previous steps are repeated for every output port of every machine $\mathsf{M}_{\mathsf{sync}} \in \kappa(j)$.

After this detailed description of the run, (i.e., the description of blocks) the mapping ϕ can be defined. Informally, it combines the blocks of all machines $\mathsf{M}_{\mathsf{sync}} \in \kappa(j)$ yielding the synchronous steps of every machine $\mathsf{M}_{\mathsf{sync}}$.

Definition 3.12 *(Mapping ϕ) Let an arbitrary synchronous system Sys_{sync} with a clocking scheme κ and an arbitrary configuration $conf_{\mathsf{sync}} = (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}}) \in \mathsf{Conf}(Sys_{\mathsf{sync}})$ be given. For a given asynchronous configuration $conf_{\mathsf{async}}$ which fits the form $conf_{\mathsf{async}} = (\varphi(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{X}_{\mathsf{sync},\kappa}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \mathsf{A}')$, we define the mapping ϕ on the runs of $conf_{\mathsf{async}}$ by the following algorithm. The algorithm has internal arrays $(inputs_{\mathsf{M},\mathsf{p}?})$ for $\mathsf{M} \in \varphi(\hat{M}_{\mathsf{sync}}) \cup \{\varphi(\mathsf{H}_{\mathsf{sync}}), \mathsf{A}'\}$ and $\mathsf{p}? \in \mathsf{in}(\mathsf{ports}(\mathsf{M}))$. It goes from block to block modifying them as follows.*

- *Every step of a buffer is deleted from the run.*
- *The two remaining steps of the first block are modified as follows. Let the clocked machine be $\varphi(\mathsf{M}_{\mathsf{sync}})$. If $\varphi(\mathsf{M}_{\mathsf{sync}}) \neq \mathsf{A}'$, the block is then replaced by $(\mathsf{M}_{\mathsf{sync}}, i, j, s, inputs_{\mathsf{M}_{\mathsf{sync}}}, s', \delta_{\mathsf{M}_{\mathsf{sync}}}(inputs_{\mathsf{M}_{\mathsf{sync}}}))$. If A' is clocked, the block is replaced by $(\mathsf{A}', i, j, s, inputs_{\mathsf{A}_{\mathsf{sync}}}, s', \mathcal{O}_{\mathsf{A}'})$. Here, s denotes the state of A' as it it switched by $\mathsf{X}_{\mathsf{sync},\kappa}$, and s' and $\mathcal{O}_{\mathsf{A}'}$ are the state and the output of the last step of the block, respectively (In case of divergation, the algorithm for defining the mapping ϕ diverges, too.).*
- *The algorithm starts searching the second block doing the following. If a machine M' receives a message i at $\mathsf{p}?$ in the second block, i is concatenated to the array $inputs_{\mathsf{M}',\mathsf{p}?}$.*
- *Finally, every step of the second block is deleted from the run.*

◇

Note that all necessary information (e.g., $\mathsf{M}_{\mathsf{sync}}, i, j, s, s'$ etc.) is already given by the stored arrays and by the block. Moreover, the new blocks built by the mapping ϕ in one particular subround do not depend on the second block of this subround. The mapping ϕ is obviously also defined on the view of arbitrary subsets of machines, because the step in the first block, carrying the information of the step, and the message receiving steps in the second block will also be part of the view of the considered machine. Furthermore, note that the mapping ϕ is explicitly defined for arbitrary adversaries A' (not only for $\varphi(\mathsf{A}_{\mathsf{sync}})$) which we will need in Theorem 3.5. Moreover, we can state the following lemma:

Lemma 3.4 *Let an arbitrary synchronous system Sys_{sync} with a clocking scheme κ be given and consider an arbitrary synchronous configuration $conf_{\mathsf{sync}} = (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}}) \in \mathsf{Conf}(Sys_{\mathsf{sync}})$ and an asynchronous configuration $conf_{\mathsf{async}} = (\varphi(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{X}_{\mathsf{sync},\kappa}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \mathsf{A}')$, If we now have given $\phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$ then we can 'reverse' the function ϕ, i.e., we can define a function ϕ_{H}^{-1} on the runs of the synchronous configuration, such that*

$$view_{conf_{\mathsf{async}}}(\varphi(\mathsf{H}_{\mathsf{sync}})) = \phi_{\mathsf{H}}^{-1}\left(\phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{H}_{\mathsf{sync}})))\right)$$

holds. If $conf_{\mathsf{async}}$ is polynomial-time, then ϕ_{H}^{-1} is polynomial-time computable. □

Proof. In order to prove the claim, we present an algorithm which undoes the changes of the algorithm for deriving the mapping ϕ: It has an internal list over Σ^+ initially empty, which will be used to construct the desired view. For every subround j, it goes through all tuples $(\mathsf{M}_{\mathsf{sync}}, i, j, s, \mathcal{I}, s', \mathcal{O}')$ modifying them as follows: If $\mathsf{M}_{\mathsf{sync}} = \mathsf{H}_{\mathsf{sync}}$ for one machine of this subround, it appends $(\varphi(\mathsf{H}_{\mathsf{sync}}), s, \mathcal{I}_{\mathsf{p}_{\mathsf{H}_{\mathsf{sync}}}?=(i,j)}, s', \mathcal{O}')$ to its internal list. Note that this tuple precisely matches the original asynchronous tuple for switching the honest user $\varphi(\mathsf{H}_{\mathsf{sync}})$ by the master scheduler. After that, it proceed through all tuples of this subround in precisely the same order they have been scheduled by the master scheduler (the algorithm is surely allowed to know the clocking scheme). For a given tuple of the form $(\mathsf{M}_{\mathsf{sync}}, i, j, s, \mathcal{I}, s', \mathcal{O}')$, it checks, whether there is a non-empty output at a port $\mathsf{p}!$ in $\mathcal{O}'$ with $\mathsf{p}? \in \mathsf{ports}(\varphi(\mathsf{H}_{\mathsf{sync}}))$. In this case, the honest user would be clocked in the second asynchronous block, so we use the state transition function $\delta_{\varphi(\mathsf{H}_{\mathsf{sync}})}$ on the current state s of $\varphi(\mathsf{H}_{\mathsf{sync}})$ and input $\mathcal{I}_{\mathsf{p}?=\mathcal{O}'_{\mathsf{p}!}}$ which yields a new state s' and an (all-empty) output $\mathcal{O}_\epsilon$. We then add a step $(\varphi(\mathsf{H}_{\mathsf{sync}}), s, \mathcal{I}_{\mathsf{p}?=\mathcal{O}'_{\mathsf{p}!}}, s', \mathcal{O}_\epsilon)$. This is done for all ports of $\mathsf{M}_{\mathsf{sync}}$ according to their order and for all machines that switch in the consider subround. Obviously, this algorithm reverses the mapping ϕ for the honest user by construction. In case of a polynomial configuration, especially the adversary has to be polynomial-time. This implies that there cannot be any infinite successive clocked self-loops. Moreover, both the adversary and the honest user will reach final state after a polynomial number of blocks, so the algorithm for ϕ_{H}^{-1} applied on the view of the honest user will only makes a polynomial number of transition, each one with a polynomial number of steps. This implies that ϕ is computable polynomial-time applied on the view of the honest user if it is used in a polynomial-time configuration. ■

Theorem 3.4 *Let an arbitrary synchronous system Sys_{sync}, a clocking scheme κ, and an arbitrary configuration $conf_{\mathsf{sync}} = (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}}) \in \mathsf{Conf}(Sys_{\mathsf{sync}})$*

be given. For brevity, set $Sys_{\mathsf{async}} := \varphi(Sys_{\mathsf{sync}})$ *and* $conf_{\mathsf{async}} := \varphi(conf_{\mathsf{sync}}) \in \mathsf{Conf}(Sys_{\mathsf{async}})$. *Then,*

$$view_{conf_{\mathsf{sync}}}(\mathsf{M}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{M}_{\mathsf{sync}})))$$

holds for every $\mathsf{M}_{\mathsf{sync}} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}}\})$. *As a special case, we obtain*

$$view_{conf_{\mathsf{sync}}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}})}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$

Moreover, $conf_{\mathsf{async}}$ *is polynomial-time iff* $conf_{\mathsf{sync}}$ *is polynomial-time.* □

Proof. Note that the view of $\varphi(\mathsf{M}_{\mathsf{sync}})$ does only contain the steps of its internal blackbox function-call after being modified by the mapping ϕ. Thus, it is sufficient to show that the inputs of the blackbox call in $conf_{\mathsf{async}}$ and the original inputs of $\mathsf{H}_{\mathsf{sync}}$ in $conf_{\mathsf{sync}}$ are equal. It is quite easy to see that the arrays $input\_store_{\mathsf{M}_{\mathsf{sync}}}$ and $inputs_{\mathsf{M}_{\mathsf{sync}}}$ are always equal if the machine $\mathsf{M}_{\mathsf{sync}}$ is switched. This can easily be proven by induction over the number of rounds. In the first round, both arrays are empty yielding a correct start of the induction. Starting with the second round, the contents of these arrays are totally determined by the inputs at the ports of $\mathsf{M}_{\mathsf{sync}}$. However, these inputs only depend on *prior* outputs of other machines M. Moreover, these outputs have to equal because these machines used the same input tuple in both configurations, since we have $input\_store_M = inputs_M$ by induction hypothesis. Therefore, the arrays $inputs_{\mathsf{M}_{\mathsf{sync}}}$ and $input\_store_{\mathsf{M}_{\mathsf{sync}}}$ must be equal at replacing the block by construction of the algorithm, so $\delta_{\mathsf{M}_{\mathsf{sync}}}(s, inputs_{\mathsf{M}_{\mathsf{sync}}}) = \delta_{\mathsf{M}_{\mathsf{sync}}}(s, input\_store_{\mathsf{M}_{\mathsf{sync}}})$ also holds. We do not have to worry about the arrangement of the blocks because of the following reasons. First of all, note that we first switch all machines in a subround and schedule the outgoing messages afterwards. Moreover, messages sent by the adversary are always scheduled first if the adversary is scheduled in the considered subround. This prevents that machines which should switch simultaneously in the synchronous system may influence each other in the asynchronous system in the same subround. If we would not consider this restriction, the adversary would be able to create a message that is scheduled in this particular subround, but nevertheless depends on inputs arriving in this subround.

Putting it all together, the runs induced by the mapping ϕ in $conf_{\mathsf{async}}$ and the original runs are equal by definition of ϕ, so we finally obtain

$$view_{conf_{\mathsf{sync}}}(\mathsf{M}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{M}_{\mathsf{sync}})))$$

for an arbitrary configuration $conf_{\mathsf{sync}} \in \mathsf{Conf}(Sys_{\mathsf{sync}})$, $conf_{\mathsf{async}} := \varphi(conf_{\mathsf{sync}})$, and an arbitrary $\mathsf{M}_{\mathsf{sync}} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}}\})$. As a special case, this implies

$$view_{conf_{\mathsf{sync}}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$$

which finishes our proof.

■

Remark 3.4. In case of a polynomial configuration, especially the adversary has to be polynomial-time. This implies that there cannot be any infinite successive clocked self-loops, so the steps of every sub-block are bounded by a polynomial in the security parameter k. Moreover, both the adversary and the honest user will reach final state after a polynomial number of blocks, so the algorithm for ϕ applied on the view either

of the honest user or the adversary only makes a polynomial number of transition, each one with a polynomial number of steps.[12] This implies that ϕ is computable polynomial-time applied on the view of either the honest user or the adversary if it is used in a polynomial-time configuration. ∘

Note, that Theorem 3.4 captures the first step of our proofsketch. After this first step, asynchronous simulatability can now be applied. In order to convert the derived asynchronous configuration into a synchronous configuration again (cf. Step 3 of our proofsketch), we present the following theorem.

Theorem 3.5 *Let an arbitrary synchronous system Sys_{sync} and a clocking scheme κ be given such that every machine and the honest user are clocked at most once between two successive clockings of the adversary.*[13] *Furthermore, let an arbitrary configuration $conf_{\mathsf{async}} \in \mathsf{Conf}(\varphi(Sys_{\mathsf{sync}}))$ of the form $conf_{\mathsf{async}} = (\varphi(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{X}_{\mathsf{sync},\kappa}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \mathsf{A}_{\mathsf{async}})$ be given and at least one of the three modifications considered at the start of this section should be made.*

Then there is an adversary $\mathsf{A}_{\mathsf{sync}}$ using $\mathsf{A}_{\mathsf{async}}$ as a blackbox such that $conf_{\mathsf{sync}} := (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}})$ yields

$$view_{conf_{\mathsf{sync}}}(\mathsf{M}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{M}_{\mathsf{sync}})))$$

for every $\mathsf{M}_{\mathsf{sync}} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}\})$ and

$$view_{conf_{\mathsf{sync}}}(\mathsf{A}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}}}(\mathsf{A}_{\mathsf{async}}))).$$

As a special case, we have

$$view_{conf_{\mathsf{sync}}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$

$conf_{\mathsf{async}}$ is polynomial-time iff $conf_{\mathsf{sync}}$ is polynomial-time.

□

Proof. The proof of this theorem is technical and tedious, so we postpone it to the Appendix. ■

3.4.4 The Embedding Theorems

We now state our first main theorem.

Theorem 3.6 *(First Embedding Theorem) Let two arbitrary synchronous systems $Sys_{\mathsf{sync},1}$ and $Sys_{\mathsf{sync},2}$ with clocking schemes κ_1 and κ_2 be given such that κ_2 fulfills the property that every machine of the system and the user is clocked at most once between two successive clockings of the adversary. Furthermore, $\varphi(Sys_{\mathsf{sync},1}) \geq^{f}_{\mathsf{async}} \varphi(Sys_{\mathsf{sync},2})$ should hold for a valid mapping f. Then*

$$Sys_{\mathsf{sync},1} \geq^{f'}_{\mathsf{sync}} Sys_{\mathsf{sync},2},$$

where f' is derived from f by $(\hat{M}_2, S_2) \in f'(\hat{M}_1, S_1) :\Leftrightarrow \varphi(\hat{M}_2, S_2) \in f(\varphi(\hat{M}_1, S_1))$.
□

[12] Deleting the steps of the buffers of one block needs a constant number of steps, because it is always bounded by the number of output ports of the considered machine, replacing the block can surely be done using a constant number of steps. Finally, searching and deleting the second block needs a polynomial number of steps.

[13] Note, that the standard clocking scheme $(\hat{M} \cup \{\mathsf{H}\}, \{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{A}\})$ fulfi lls this precondition.

Using the result of the previous theorems, the proof will be rather simple; it is illustrated in Figure 3.6. Since we already presented an outline of the proof at the start of the section, we omit the usual proofsketch this time.

Proof. Let an arbitrary configuration $conf_{\mathsf{sync},1} = (\hat{M}_{\mathsf{sync},1}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync},1}) \in \mathsf{Conf}(Sys_{\mathsf{sync},1})$ be given.

1. We apply φ_{conf} on $conf_{\mathsf{sync},1}$ yielding a configuration $conf_{\mathsf{async},1} = (\varphi(\hat{M}_{\mathsf{sync},1}) \cup \{\mathsf{X}_{\mathsf{sync},1,\kappa_1}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \varphi(\mathsf{A}_{\mathsf{sync},1})) \in \mathsf{Conf}(Sys_{\mathsf{async},1})$. According to Theorem 3.4, there is a mapping ϕ on the runs of $conf_{\mathsf{async},1}$ such that

   $$view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$$

   holds. Moreover, if $conf_{\mathsf{sync},1}$ is polynomial-time then $conf_{\mathsf{async},1}$ is also polynomial-time, and the mapping ϕ is polynomial-time computable.

2. As usual, we can assume $conf_{\mathsf{async},1}$ to be suitable without loss of generality. Thus, the precondition $\varphi(Sys_{\mathsf{sync},1}) \geq_{\mathsf{async}} \varphi(Sys_{\mathsf{sync},2})$ can be applied yielding a configuration $conf_{\mathsf{async},2} = (\varphi(\hat{M}_{\mathsf{sync},2}) \cup \{\mathsf{X}_{\mathsf{sync},2,\kappa_2}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \mathsf{A}_{\mathsf{async},2}) \in \mathsf{Conf}(Sys_{\mathsf{async},2})$ with

   $$view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})) \approx view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}})).$$

   Moreover, in the computational case, $conf_{\mathsf{async},2}$ is polynomial-time, so the mapping ϕ is polynomial-time computable. Using Lemma 2.4, this yields

   $$\phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) \approx \phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$

3. We want to apply Theorem 3.5 to the configuration $conf_{\mathsf{async},2}$, so we have to show that all preconditions are fulfilled which obviously holds for $conf_{\mathsf{async},2}$. Thus, Theorem 3.5 yields a configuration $conf_{\mathsf{sync},2} = (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync},2})$ with

   $$\phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) = view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}}).$$

   Moreover, $conf_{\mathsf{sync},2}$ is a polynomial configuration iff $conf_{\mathsf{async},2}$ is polynomial, according to Theorem 3.5.

4. Putting it all together, we have

   - $view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$
   - $\phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) \approx \phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$
   - $\phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) = view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})$

   Using Lemma 2.4, we obtain $view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) \approx view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})$. Hence, $conf_{\mathsf{sync},2}$ is an indistinguishable configuration for $conf_{\mathsf{sync},1}$ which yields the desired result $Sys_{\mathsf{sync},1} \geq_{\mathsf{sync}} Sys_{\mathsf{sync},2}$.

■

Corollary 3.1 *Let two arbitrary synchronous systems $Sys_{\mathsf{sync},1}$ and $Sys_{\mathsf{sync},2}$ and a valid mapping f be given. Moreover, let $\kappa := \kappa_1 := \kappa_2$ be the standard clocking scheme $(\hat{M} \cup \{\mathsf{H}\}, \{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{A}\})$. Then $\varphi(Sys_{\mathsf{sync},1}) \geq^{f}_{\mathsf{async}} \varphi(Sys_{\mathsf{sync},2})$ implies $Sys_{\mathsf{sync},1} \geq^{f'}_{\mathsf{sync}} Sys_{\mathsf{sync},2}$ with f' defined as in Theorem 3.6.* □

This is a direct consequence of the previous theorem because the standard clocking scheme obviously fulfills the requirement that no machine is clocked more than once between two successive clockings of the adversary.

Putting it all together, we have shown so far that asynchronous simulatability among these asynchronous representations implies synchronous simulatability in the computational case, and in the perfect and statistical case under some reasonable assumptions, i.e., we have shown

$$\varphi_{Sys}(Sys_{\mathsf{sync},1}) \geq_{\mathsf{async}} \varphi_{Sys}(Sys_{\mathsf{sync},2}) \Rightarrow Sys_{\mathsf{sync},1} \geq_{\mathsf{sync}} Sys_{\mathsf{sync},2}.$$

We already briefly stated above that the converse implication does not hold in general. We would have to show that for an arbitrary configuration $conf_{\mathsf{async},1} \in \mathsf{Conf}(\varphi_{Sys}(Sys_{\mathsf{sync},1}))$ there exists an indistinguishable configuration $conf_{\mathsf{async},2} \in \mathsf{Conf}(\varphi_{Sys}(Sys_{\mathsf{sync},2}))$ provided that $Sys_{\mathsf{sync},1} \geq_{\mathsf{sync}} Sys_{\mathsf{sync},2}$ holds.

However, we cannot expect that to hold if both the honest user and the adversary are completely unrestricted. Obviously, both machines may have clockout ports and they can alternately schedule each other (and also the system erratically), which we cannot capture by a synchronous clocking scheme, so we cannot exploit our assumption $Sys_{\mathsf{sync},1} \geq_{\mathsf{sync}} Sys_{\mathsf{sync},2}$.

Anyhow, it is sufficient for our purpose to show that the claim holds for at least those configurations where the honest user $\mathsf{H}_{\mathsf{async}}$ fits the form $\varphi_{\mathsf{M}}(\mathsf{H}_{\mathsf{sync}})$ for a synchronous machine $\mathsf{H}_{\mathsf{sync}}$. We denote this restricted version of simulatability by $\geq_{\mathsf{async,H}}$ in the following.

Definition 3.13 *($\geq_{\mathsf{async,H}}$) Let two arbitrary asynchronous systems Sys_1 and Sys_2 and a valid mapping f be given. Then we say $Sys_1 \geq^{f}_{\mathsf{async,H}} Sys_2$ if for every configuration $conf_1 = (\hat{M}_1, S, \mathsf{H}_{\mathsf{async}}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1)$ with $\mathsf{H}_{\mathsf{async}} = \varphi_{\mathsf{M}}(\mathsf{H}_{\mathsf{sync}})$ for a synchronous machine $\mathsf{H}_{\mathsf{sync}}$ there exists an indistinguishable configuration $conf_2 = (\hat{M}_2, S, \mathsf{H}_{\mathsf{async}}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$ with $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$.* ◇

Remark 3.5. Looking at the proof of the first embedding theorem 3.6, it is immediately obvious that the theorem also holds for the weaker precondition $\varphi_{Sys}(Sys_{\mathsf{sync},1}) \geq_{\mathsf{async,H}} \varphi_{Sys}(Sys_{\mathsf{sync},2})$, since we only need to derive an indistinguishable configuration for users of the special form $\varphi(\mathsf{H}_{\mathsf{sync}})$. Moreover, the user remains unchanged at simulatability, so we obtain a configuration of the same user again, which still fits the prescribed form. ∘

If we restrict our attention to this kind of simulatability, we can state the following claim:

$$Sys_{\mathsf{sync},1} \geq_{\mathsf{sync}} Sys_{\mathsf{sync},2} \Rightarrow \varphi_{Sys}(Sys_{\mathsf{sync},1}) \geq_{\mathsf{async,H}} \varphi_{Sys}(Sys_{\mathsf{sync},2}),$$

which captures the statement of the second embedding theorem.

Theorem 3.7 *(Second Embedding Theorem) Let two arbitrary synchronous systems $Sys_{\mathsf{sync},1}$ and $Sys_{\mathsf{sync},2}$ with clocking schemes κ_1 and κ_2 be given such that κ_1 fulfills the property that every machine of the system and the user is clocked at most once between two successive clockings of the adversary. Furthermore, $Sys_{\mathsf{sync},1} \geq^{f}_{\mathsf{sync}} Sys_{\mathsf{sync},2}$ should hold for a valid mapping f. Then*

$$\varphi(Sys_{\mathsf{sync},1}) \geq^{f'}_{\mathsf{async,H}} \varphi(Sys_{\mathsf{sync},2})$$

where f' is derived from f by $\varphi(\hat{M}_2, S_2) \in f'(\varphi(\hat{M}_1, S_1)) :\Leftrightarrow (\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$. □

Using the result of the preliminary theorems, the proof will be rather simple. Since it will be performed similar to the first embedding theorem, we omit the usual proofsketch this time.

Proof. For readability, we again set $Sys_{\mathsf{async},1} := \varphi(Sys_{\mathsf{sync},1})$ and $Sys_{\mathsf{async},2} := \varphi(Sys_{\mathsf{sync},2})$. Let now an arbitrary configuration $conf_{\mathsf{async},1} = (\varphi(\hat{M}_{\mathsf{sync},1}) \cup \{\mathsf{X}_{\mathsf{sync},1,\kappa_1}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \mathsf{A}_{\mathsf{async},1}) \in \mathsf{Conf}(Sys_{\mathsf{async},1})$ be given.

1. We apply theorem 3.5 on $conf_{\mathsf{async},1}$ which yields a synchronous configuration $conf_{\mathsf{sync},1} = (\hat{M}_{\mathsf{sync},1}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync},1}) \in \mathsf{Conf}(Sys_{\mathsf{sync},1})$ with

   $$\phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) = view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}).$$

   Moreover, if $conf_{\mathsf{async},1}$ is polynomial-time then $conf_{\mathsf{sync},1}$ is also polynomial-time, and the mapping ϕ is polynomial-time computable.

2. As usual, we can assume $conf_{\mathsf{sync},1}$ to be suitable without loss of generality. Thus, the precondition $Sys_{\mathsf{sync},1} \geq_{\mathsf{sync}} Sys_{\mathsf{sync},2}$ can be applied yielding a configuration $conf_{\mathsf{sync},2} = (\hat{M}_{\mathsf{sync},2}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync},2}) \in \mathsf{Conf}(Sys_{\mathsf{sync},2})$ with

   $$view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) \approx view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}}).$$

   Moreover, in the computational case, $conf_{\mathsf{async},2}$ is polynomial-time.

3. We now apply Theorem 3.4 to the configuration $conf_{\mathsf{sync},2}$ which yields a configuration $conf_{\mathsf{async},2} = (\varphi(\hat{M}_{\mathsf{sync},2}) \cup \{\mathsf{X}_{\mathsf{sync},2,\kappa_2}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \varphi(\mathsf{A}_{\mathsf{sync},2}))$ with

   $$view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$

   Moreover, $conf_{\mathsf{async},2}$ is a polynomial configuration iff $conf_{\mathsf{sync},2}$ is polynomial, according to Theorem 3.4.

4. Putting it all together, we have

   - $\phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) = view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}})$
   - $view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) \approx view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})$
   - $view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})) = \phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$

   Using Lemma 2.4, we obtain

   $$\phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) \approx \phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$

   We now finally apply our "reversing" function ϕ_{H}^{-1} (cf. Lemma 3.4) on the above equation which yields

   $$view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})) \approx view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}})).$$

   Hence, $conf_{\mathsf{async},2}$ is an indistinguishable configuration for $conf_{\mathsf{async},1}$ which yields the desired result $\varphi(Sys_{\mathsf{sync},1}) \geq_{\mathsf{async},\mathsf{H}} \varphi(Sys_{\mathsf{async},2})$.

■

3.4.5 An application

Recall that our goal is to avoid proving each and every theorem and lemma for both models. We now briefly sketch how our two embedding theorems can be used for circumventing this problem. As an example, we will show that it would have been sufficient to prove the asynchronous version of the transitivity lemma 2.6, because the synchronous version automatically follows. This is captured in the following lemma.

Lemma 3.5 *(Asynchronous Version of Transitivity implies Synchronous Version) Assume that the asynchronous version of the transitivity lemma 2.6 has already been proven, then the synchronous version holds as well.* □

Proof. Let three arbitrary synchronous systems Sys_1, Sys_2, and Sys_3 be given such that $Sys_1 \geq_{\mathsf{sync}} Sys_2$ and $Sys_2 \geq_{\mathsf{sync}} Sys_3$. We have to show that $Sys_1 \geq_{\mathsf{sync}} Sys_3$ holds, provided that asynchronous transitivity has already been proven. According to our second embedding theorem, we know that

$$\varphi(Sys_1) \geq_{\mathsf{async,H}} \varphi(Sys_2)$$

and

$$\varphi(Sys_2) \geq_{\mathsf{async,H}} \varphi(Sys_3)$$

hold. Obviously, the asynchronous version of transitivity is applicable to the relation $\geq_{\mathsf{async,H}}$ instead of $\geq_{\mathsf{async}}$ as well, since it is a special case only, and the honest user remains unchanged at simulatability. Thus, we can apply our (already proven) asynchronous version of the transitivity lemma, which yields

$$\varphi(Sys_1) \geq_{\mathsf{async,H}} \varphi(Sys_3).$$

Now, we use our first embedding theorem in conjunction with Remark 3.5 which yields $Sys_1 \geq_{\mathsf{sync}} Sys_3$. ■

Chapter 4

Deriving Secure Implementations

Starting with this chapter, we deal with the actual verification of cryptographic protocols. We present a monolithic specification of secure message transmission with ordered channels and a concrete, secure implementation. The way of actually deriving the implementation comprises a general methodology how concrete implementations of abstract specifications can be found. Our approach for proving security is essentially based on formally verified bisimulations, which yield trustworthy proofs. After accomplishing some preparatory work for proving the security of our implementation, we formally verified its security using the theorem prover PVS [44]. We conclude this chapter with a brief summary of its results.

Prior to this work, there has not been any success in using the advantage of formal verification in order to derive cryptographically sound implementations, so our methodology is new, and it serves as our first step of bridging the gap between the rigorous proofs of cryptography and verification using formal proof systems.

4.1 Secure Message Transmission in Correct Order

In this section an abstract specification for *secure message transmission with ordered channels* is presented, so neither reordering the messages in transit nor replay attacks are possible for the adversary. Furthermore, we present a real system that fulfills our specification, so it can be regarded as a possible implementation.

The section is structured as follows. First, we present our abstract specification; after that we split it into two submachines and apply the composition theorem yielding a possible implementation. The security of this implementation is additionally based on the handmade proof of Section 2.3. In the following two sections, we prove this real system to be at least as secure as the specification. This successfully finishes our attempt to design a verified, concrete system fulfilling our specification.

4.1.1 The Abstract Specification

Our specification is a standard ideal system $Sys^{\mathsf{spec}} = \{(\mathsf{TH}'_{\mathcal{H}}, S_{\mathcal{H}}) | \mathcal{H} \in \mathcal{ACC}\}$ as described in Section 2.2.1 where any number of participants may be dishonest. We start with some intuitive description of how the scheme works.

The ideal machine $\mathsf{TH}'_{\mathcal{H}}$ basically acts similar to the trusted honest $\mathsf{TH}_{\mathcal{H}}$ of the ideal scheme for secure message transmission (cf. Scheme 2.1) with some minor, but far-reaching changes.

As in Scheme 2.1, a user u can initialize communications with other users by inputting a command of the form $(\mathsf{snd\_init})$ to the port $\mathsf{in}_u?$ of $\mathsf{TH}'_{\mathcal{H}}$, sending of messages to a user v is triggered by a command (send, m, v).

The "Send" transition of $\mathsf{TH}'_{\mathcal{H}}$ is modified: if v is honest, the message is stored in an internal array $deliver^{\mathsf{spec}}_{u,v}$ of $\mathsf{TH}'_{\mathcal{H}}$ together with a counter $msg\_in^{\mathsf{spec}}_{u,v}$ indicating the number of the message. This counter is the first essential difference between the two systems which will allow us to identify messages which are either replayed or out-of-order.

Just as in the original scheme, a message $(\mathsf{send\_blindly}, i, l, v)$ is output to the adversary, where l and i denote the length of the message m and its position in the array, respectively. Because of the underlying asynchronous timing model, $\mathsf{TH}'_{\mathcal{H}}$ has to wait for a special term $(\mathsf{receive\_blindly}, u, i)$ or $(\mathsf{rec\_init}, u)$ sent by the adversary at $\mathsf{from\_adv}_v!$, signaling that the message stored at the ith position of $deliver^{\mathsf{spec}}_{u,v}$ should be delivered to v or that a connection between u and v should be initialized, respectively.

In the first case, the machine $\mathsf{TH}'_{\mathcal{H}}$ has to ensure that messages will only pass if their order has not been changed, i.e., we must prevent replay attacks and message reordering. Thus, $\mathsf{TH}'_{\mathcal{H}}$ reads $(m, j) := deliver^{\mathsf{spec}}_{u,v}[i]$ and checks whether $msg\_out^{\mathsf{spec}}_{u,v} \leq j$ holds for a message counter $msg\_out^{\mathsf{spec}}_{u,v}$. If this test is successful the message is delivered and the counter is set to $j + 1$. Otherwise, $\mathsf{TH}'_{\mathcal{H}}$ outputs nothing. This check and the following increasing of the counter will prevent both message reordering and replay attacks. Informally, the variable $msg\_out^{\mathsf{spec}}_{u,v}$ stores the number of the next delivered message from u to v. If now a message m sent from u to v should be delivered whose number j is less than the internal counter $msg\_out^{\mathsf{spec}}_{u,v}$, there has to be a message with higher number that has already been delivered, so we either have a message reordering or a replay attack. Thus, $\mathsf{TH}'_{\mathcal{H}}$ does not output anything in this case. The user will receive inputs of the form $(\mathsf{receive}, u, m)$ and $(\mathsf{rec\_init}, u)$, respectively.

If v is dishonest, $\mathsf{TH}'_{\mathcal{H}}$ will simply output (send, m, v) to the adversary. Finally, the adversary can send a message m to a user u by sending a command $(\mathsf{receive}, v, m)$ to the port $\mathsf{from\_adv}_u?$ of $\mathsf{TH}'_{\mathcal{H}}$ for a corrupted user v, and he can also stop the machine of any user by sending a command (stop) to $\mathsf{TH}'_{\mathcal{H}}$ which corresponds to exceeding the machine's runtime bounds in the real world.

The length of each message and the number of messages each user may send and receive are bounded by $L(k)$, $s_1(k)$ and $s_2(k)$, respectively, for polynomials L, s_1, s_2, and the security parameter k. We will furthermore distinguish between the *standard ordered system* and the *perfect ordered system*. The standard ordered system only prevents message reordering, but the adversary can still leave out messages. In the perfect ordered system, the adversary is just able to deliver messages between honest users in exactly the sequence they have been sent. We now give the specification of the system.

Scheme 4.1 (Specification for Ordered Secure Message Transmission) Let $n \in \mathbb{N}$ and polynomials $L, s_1, s_2 \in \mathbb{N}[x]$ be given. Let $\mathcal{M} := \{1, \ldots, n\}$ denote the set of possible participants, and let the access structure $\mathcal{ACC}$ be the powerset of $\mathcal{M}$. Our specification for secure message transmission with ordered channels is a standard ideal system

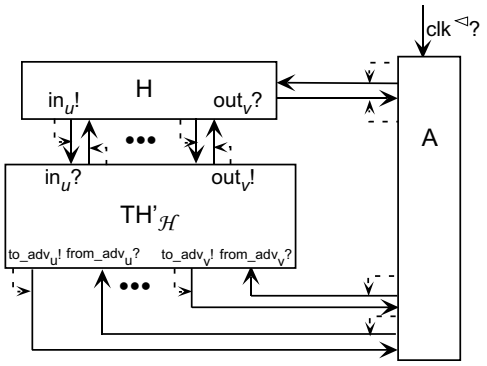

Figure 4.1: Ports in a Configuration of Secure Message Transmission with Ordered Channels.

$$Sys^{\mathsf{msg\_ord,spec}}_{n,L,s_1,s_2} = \{(\{\mathsf{TH}'_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \subseteq \mathcal{M}\}$$

with the standard localized definition $S^c_{\mathcal{H}} := \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$ and $\mathsf{TH}'_{\mathcal{H}}$ defined as follows. When $\mathcal{H}$ is clear from the context, let $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$ denote the indices of corrupted participants.

The ports of the machine $\mathsf{TH}'_{\mathcal{H}}$ are $\{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{\mathsf{from\_adv}_u?, \mathsf{to\_adv}_u!, \mathsf{to\_adv}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$. Internally, $\mathsf{TH}'_{\mathcal{H}}$ maintains seven arrays:

- $(init^{\mathsf{spec}}_{u,v})_{u,v\in\mathcal{M}}$ over $\{0,1\}$ for modeling initialization of users,
- $(sc\_in^{\mathsf{spec}}_{u,v})_{u\in\mathcal{H},v\in\mathcal{M}}$ over $\{0,\ldots,s_1(k)\}$ for counting the number of times $\mathsf{TH}'_{\mathcal{H}}$ has been switched by user u using messages intended to v,
- $(msg\_out^{\mathsf{spec}}_{u,v})_{u,v\in\mathcal{H}}$ over $\{0,\ldots,s_2(k)\}$ for counting the number of the next possible message, cf. our above description.
- $(sc\_out^{\mathsf{spec}}_{u,v})_{u\in\mathcal{M},v\in\mathcal{H}}$ over $\{0,\ldots,s_2(k)\}$ for counting the number of times $\mathsf{TH}'_{\mathcal{H}}$ has been switched by the adversary for delivering a message from user u to user v,
- $(msg\_in^{\mathsf{spec}}_{u,v})_{u\in\mathcal{H},v\in\mathcal{M}}$ over $\{0,\ldots,s_1(k)\}$ for counting the number of ingoing messages from u intended to v,
- $(stopped^{\mathsf{spec}}_{u})_{u\in\mathcal{H}}$ over $\{0,1\}$: This array stores whether the machine of user u has already been stopped, i.e., it has reached its runtime bounds,
- $(deliver^{\mathsf{spec}}_{u,v})_{u,v\in\mathcal{H}}$ of lists for storing the actual messages.

The first six arrays should be initialized with 0 everywhere, except that $msg\_out^{\mathsf{spec}}_{u,v}$ is initialized with 1 everywhere. The last array should be initialized with empty lists everywhere. Roughly, the five arrays $init^{\mathsf{spec}}_{u,v}$, $msg\_out^{\mathsf{spec}}_{u,v}$, $msg\_in^{\mathsf{spec}}_{u,v}$, $stopped^{\mathsf{spec}}_{u}$, and $deliver^{\mathsf{spec}}_{u,v}$ ensure functional correctness of the system whereas the arrays $sc\_in^{\mathsf{spec}}_{u,v}$ and $sc\_out^{\mathsf{spec}}_{u,v}$ have to be included to make the system polynomial-time. During the run of a configuration, the arrays $sc\_in^{\mathsf{spec}}_{u,v}$ and $sc\_out^{\mathsf{spec}}_{u,v}$ will be increased and the machine $\mathsf{TH}'_{\mathcal{H}}$ will check whether their values are still smaller than the given polynomial bounds $s_1(k)$ or $s_2(k)$, respectively. If at least one value is greater than this

bound, the machine $\mathsf{TH}'_{\mathcal{H}}$ stops, i.e., it enters a final state. The length bounds of the machine $\mathsf{TH}'_{\mathcal{H}}$ for incoming messages can be chosen arbitrarily as long as they are polynomially bounded. The state-transition function of $\mathsf{TH}'_{\mathcal{H}}$ is defined by the following rules, written in the usual pseudo-code language. For the sake of readability, we again annotate the first part of the definition, the "Send initialization" transition, i.e., the key generation in the real world.

Initialization.

- **Send initialization:**

  Assume, that the user u wants to generate its encryption and signature keys and distribute the corresponding public keys over authenticated channels. He can do so by sending a command $(\mathsf{snd\_init})$ to $\mathsf{TH}'_{\mathcal{H}}$. Now, the system checks that the user has not already reached his message bound (which is quite improbable in this case unless he tried to send trash all the time), that the machine itself has not reached its runtime bound, and that no key generation of this user has already occurred in the past. These three checks correspond to $sc\_in^{\mathsf{spec}}_{u,v} < s_1(k)$ for all $v \in \mathcal{M}$, $stopped^{\mathsf{spec}}_u = 0$, and $init^{\mathsf{spec}}_{u,u} = 0$, respectively. If at least the check of the message bound (i.e., $sc\_in^{\mathsf{spec}}_{u,v} < s_1(k)$) holds, the counter $sc\_in^{\mathsf{spec}}_{u,v}$ is increased. If all three checks hold, the keys are additionally distributed over authenticated channels, modeled by an output $(\mathsf{snd\_init})$ to the adversary which either can schedule them immediately, later or even leave them on the channels forever. In our pseudo-code language this is expressed as follows:

  On input $(\mathsf{snd\_init})$ at $\mathsf{in}_u?$: If $sc\_in^{\mathsf{spec}}_{u,v} < s_1(k)$ for all $v \in \mathcal{M}$, set $sc\_in^{\mathsf{spec}}_{u,v} := sc\_in^{\mathsf{spec}}_{u,v} + 1$ for all $v \in \mathcal{M}$, otherwise do nothing. If the test holds check $stopped^{\mathsf{spec}}_u = 0$ and $init^{\mathsf{spec}}_{u,u} = 0$. In this case set $init^{\mathsf{spec}}_{u,u} := 1$ and output $(\mathsf{snd\_init})$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$.

  The following parts should now be understood similarly:

- **Receive initialization:** On input $(\mathsf{rec\_init}, u)$ at $\mathsf{from\_adv}_v?$ with $u \in \mathcal{M}, v \in \mathcal{H}$: If $stopped^{\mathsf{spec}}_v = 0$, $init^{\mathsf{spec}}_{u,v} = 0$, and $[u \in \mathcal{H} \Rightarrow init^{\mathsf{spec}}_{u,u} = 1]$, set $init^{\mathsf{spec}}_{u,v} := 1$. If $sc\_out^{\mathsf{spec}}_{u,v} < s_2(k)$ set $sc\_out^{\mathsf{spec}}_{u,v} := sc\_out^{\mathsf{spec}}_{u,v} + 1$, output $(\mathsf{rec\_init}, u)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

Sending and receiving messages.

- **Send:** On input (send, m, v) at $\mathsf{in}_u?$, $v \in \mathcal{M} \setminus \{u\}$: If $sc\_in^{\mathsf{spec}}_{u,v} < s_1(k)$ and $stopped^{\mathsf{spec}}_u = 0$, set $sc\_in^{\mathsf{spec}}_{u,v} := sc\_in^{\mathsf{spec}}_{u,v} + 1$, otherwise do nothing. If $m \in \Sigma^+$, $l := \mathsf{len}(m) \leq L(k)$, $init^{\mathsf{spec}}_{u,u} = 1$ and $init^{\mathsf{spec}}_{v,u} = 1$ holds:

  If $v \in \mathcal{A}$ then $\{$ set $msg\_in^{\mathsf{spec}}_{u,v} := msg\_in^{\mathsf{spec}}_{u,v} + 1$ and output $(\mathsf{send}, (m, msg\_in^{\mathsf{spec}}_{u,v}), v)$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$ $\}$ else $\{$set $i := \mathsf{size}(deliver^{\mathsf{spec}}_{u,v}) + 1$, $msg\_in^{\mathsf{spec}}_{u,v} := msg\_in^{\mathsf{spec}}_{u,v} + 1$, $deliver^{\mathsf{spec}}_{u,v}[i] := (m, msg\_in^{\mathsf{spec}}_{u,v})$ and output $(\mathsf{send\_blindy}, i, l, v)$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$ $\}$.

- **Receive from honest party u:** On input $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}_v?$ with $u, v \in \mathcal{H}$: If $stopped^{\mathsf{spec}}_v = 0$, $init^{\mathsf{spec}}_{v,v} = 1$, $init^{\mathsf{spec}}_{u,v} = 1$, $sc\_out^{\mathsf{spec}}_{u,v} < s_2(k)$ and $(m, j) := deliver^{\mathsf{spec}}_{u,v}[i] \neq \downarrow$, check $msg\_out^{\mathsf{spec}}_{u,v} \leq j$ ($msg\_out^{\mathsf{spec}}_{u,v} = j$ in the perfect ordered system). If this holds set $sc\_out^{\mathsf{spec}}_{u,v} := sc\_out^{\mathsf{spec}}_{u,v} + 1$, $msg\_out^{\mathsf{spec}}_{u,v} := j + 1$ and output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

- **Receive from dishonest party u:** On input $(\mathsf{receive}, u, m)$ at $\mathsf{from\_adv}_v?$ with $u \in \mathcal{A}, m \in \Sigma^+, \mathsf{len}(m) \leq L(k)$ and $v \in \mathcal{H}$: If $stopped_v^{\mathsf{spec}} = 0$, $init_{v,v}^{\mathsf{spec}} = 1$, $init_{u,v}^{\mathsf{spec}} = 1$ and $sc\_out_{u,v}^{\mathsf{spec}} < s_2(k)$, set $sc\_out_{u,v}^{\mathsf{spec}} := sc\_out_{u,v}^{\mathsf{spec}} + 1$ and output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

- **Stop:** On input (stop) at $\mathsf{from\_adv}_u?$ with $u \in \mathcal{H}$: If $stopped_u^{\mathsf{spec}} = 0$, set $stopped_u^{\mathsf{spec}} := 1$ and output (stop) at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

Finally, if $\mathsf{TH}'_{\mathcal{H}}$ receives an input at a port $\mathsf{in}_u?$ which is not comprised by the above six transition (i.e., the user sends some kind of trash), it increases the counter $sc\_in_{u,v}^{\mathsf{spec}}$ for all $v \in \mathcal{M}$. Similarly, if $\mathsf{TH}'_{\mathcal{H}}$ receives such an input at a port $\mathsf{from\_adv}'_v?$ it increases every counter $sc\_out_{u,v}^{\mathsf{spec}}$ for $u \in \mathcal{H}$. ◇

Thus, at least one counter of $sc\_in_{u,c}^{\mathsf{spec}}$ or $sc\_out_{u,v}^{\mathsf{spec}}$ will be increases after each transition of the machine $\mathsf{TH}'_{\mathcal{H}}$, and each transition of $\mathsf{TH}'_{\mathcal{H}}$ can obviously be realized in polynomial-time since the length functions on the ports are polynomially bounded. There are only a finite number of counters and $\mathsf{TH}'_{\mathcal{H}}$ will enter a final state if at least one counter reaches its polynomial bounds. Thus, the machine $\mathsf{TH}'_{\mathcal{H}}$ is polynomial-time.

$Sys_{n,L,s_1,s_2}^{\mathsf{msg\_ord},\mathsf{spec}}$ is as abstract as we hoped for. It is deterministic without containing any cryptographic objects. Furthermore, its state transition function is kept simple, so we can express it quite easily in formal languages which support our used data-types (e.g., in PVS, cf. Section 4.3.1). In the following we simply write $Sys^{\mathsf{msg\_ord},\mathsf{spec}}$ instead of $Sys_{n,L,s_1,s_2}^{\mathsf{msg\_ord},\mathsf{spec}}$ if the parameters n, L, s_1, s_2 are not necessary for understanding.

4.1.2 The Split Ideal System

This section contains the first step for deriving a real system that implements our Scheme 4.1. A brief overview of how the real system will be derived can be given in two steps:

If we take a look at Figure 2.3 on page 20, the system $Sys^{\mathsf{msg\_ord},\mathsf{spec}}$ plays the role of the monolithic specification Sys^{spec}. We will now "split" our specification into a system Sys^*, and we will show that $Sys^* \geq_{\mathsf{sec}} Sys^{\mathsf{spec}}$ holds. Sys^* will consist of the combination of two systems Sys'_0 and Sys_1. Finally, we will replace Sys'_0 with Sys_0 using the composition theorem, and we obtain a real system that still fulfills our requirements.

The systems Sys'_0 and Sys_0 are the ideal and real systems Sys_{id} and Sys_{real} for secure message transmission, respectively, which we introduced in Section 2.3. The system Sys_1 will do the filtering of messages that are out of order.

Scheme 4.2 (Sys_1) Let $n, L, s_1, s_2, \mathcal{M}$ be given as in Scheme 4.1. Furthermore, let a polynomial $L_1 := L + c(k)$ be given; the value of $c(k)$ is explained below. Sys_1 is now defined as

$$Sys_1 = \{(\hat{M}'_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \subseteq \mathcal{M}\}$$

with $\hat{M}'_{\mathcal{H}} = \{\mathsf{M}'_u \mid u \in \mathcal{H}\}$ and $\mathsf{ports}(\mathsf{M}'_u) = \{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}!\} \cup \{\mathsf{in}'_u!, \mathsf{out}'_u?, \mathsf{in}'_u{}^{\triangleleft}!\}$. The set of specified ports is given by $S_{\mathcal{H}} = \mathsf{ports}([\hat{M}'_{\mathcal{H}}])$, i.e., all ports are specified. Internally, the machine M'_u maintains two arrays $(msg\_in_{u,v}^{\mathsf{id}})_{v \in \mathcal{M}}$, $(sc\_in_{u,v}^{\mathsf{id}})_{v \in \mathcal{M}}$ over $\{0, \ldots, s_1(k)\}$ and two arrays

$(msg\_out^{\mathsf{id}}_{v,u})_{v\in\mathcal{M}}$, $(sc\_out^{\mathsf{id}}_{v,u})_{v\in\mathcal{M}}$ over $\{0,\ldots,s_2(k)\}$. All four arrays are initialized with 0 everywhere. Moreover, it contains a flag $(stopped^{\mathsf{id}}_u)$ over $\{0,1\}$ initialized with 0. As message space, we assume $\Sigma \cup (\Sigma \times \mathbb{N})$. Furthermore, we assume that encoding of tuples has the following straightforward property with respect to the function len: We demand $\mathsf{len}((m,num)) = \mathsf{len}(m) + c(k)$ for every $num \in \{0,\ldots,\max\{s_1(k),s_2(k)\}\}$ and an arbitrary function c, i.e., $\mathsf{len}(num)$ has to be constant for a fixed security parameter k. Obviously, this condition can easily be achieved by padding all values num to a fixed size $\geq |\max\{s_1(k),s_2(k)\}|$. As length bounds for incoming messages of the machine M'_u we use the same bounds as for the corresponding ports of the machine $\mathsf{TH}'_{\mathcal{H}}$, i.e., they are in particular polynomially bounded. The behaviour of M'_u is defined as follows.

Initialization.

- **Send initialization:** On input $(\mathsf{snd\_init})$ at $\mathsf{in}_u?$: If $sc\_in^{\mathsf{id}}_{u,v} < s_1(k)$ for every $v \in \mathcal{M}$, set $sc\_in^{\mathsf{id}}_{u,v} := sc\_in^{\mathsf{id}}_{u,v} + 1$ for every $v \in \mathcal{M}$. If $stopped^{\mathsf{id}}_u = 0$ then output $(\mathsf{snd\_init})$ at $\mathsf{in}'_u!$, 1 at $\mathsf{in}'^{\triangleleft}_u!$.
- **Receive initialization:** On input $(\mathsf{rec\_init}, v)$ at $\mathsf{out}'_u?$: If $stopped^{\mathsf{id}}_u = 0$ and $sc\_out^{\mathsf{id}}_{v,u} < s_2(k)$, set $sc\_out^{\mathsf{id}}_{v,u} := sc\_out^{\mathsf{id}}_{v,u} + 1$ and output $(\mathsf{rec\_init}, v)$ at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

Sending and receiving messages.

- **Send:** On input (send, m, v) at $\mathsf{in}_u?$: If $stopped^{\mathsf{id}}_u = 0$ and $sc\_in^{\mathsf{id}}_{u,v} < s_1(k)$, set $sc\_in^{\mathsf{id}}_{u,v} := sc\_in^{\mathsf{id}}_{u,v} + 1$, $msg\_in^{\mathsf{id}}_{u,v} := msg\_in^{\mathsf{id}}_{u,v} + 1$ and output $(\mathsf{send}, (m, msg\_in^{\mathsf{id}}_{u,v}), v)$ at port $\mathsf{in}'_u!$, 1 at port $\mathsf{in}'^{\triangleleft}_u!$.
- **Receive:** On input $(\mathsf{receive}, v, m')$ at port $\mathsf{out}'_u?$: If $stopped^{\mathsf{id}}_u = 0$ and $sc\_out^{\mathsf{id}}_{v,u} < s_2(k)$, decompose the message m' into (m, num). If $msg\_out^{\mathsf{id}}_{v,u} \leq num$ (or $msg\_out^{\mathsf{id}}_{v,u} = num$ in the perfect ordered system), set $sc\_out^{\mathsf{id}}_{v,u} := sc\_out^{\mathsf{id}}_{v,u} + 1$, $msg\_out^{\mathsf{id}}_{v,u} := num + 1$ and output $(\mathsf{receive}, v, m)$ at port $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.[1]
- **Stop:** On input (stop) at $\mathsf{out}'_u?$: If $stopped^{\mathsf{id}}_u = 0$, set $stopped^{\mathsf{id}}_u := 1$ and output (stop) at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

Finally, if M'_u receives an input at a port $\mathsf{in}_u?$ which is not comprised by the above five transition (i.e., the user sends some kind of trash), it increases the counter $sc\_in^{\mathsf{spec}}_{u,v}$ for all $v \in \mathcal{M}$. Similarly, if M'_v receives such an input at a port $\mathsf{out}'_v?$ it increases every counter $sc\_out^{\mathsf{spec}}_{u,v}$ for $u \in \mathcal{H}$. ◇

Obviously, Sys'_0 is polynomial-time because of the same reason as $\mathsf{TH}'_{\mathcal{H}}$. Before we build the combination of Sys'_0 and Sys_1, we rename the ports $\mathsf{in}_u?$, $\mathsf{out}_u!$ and $\mathsf{out}_u{}^{\triangleleft}!$ of Sys'_0 into $\mathsf{in}'_u?$, $\mathsf{out}'_u!$ and $\mathsf{out}'^{\triangleleft}_u!$, respectively. Moreover, we assume $\Sigma \cup \Sigma \times \mathbb{N}$ as message space of Sys'_0 which ensures that incoming tuples can in fact be stored in the array $deliver^*_{u,v}$ (cf. the machine description in Section 4.3.1).

[1] Note that also messages sent by the adversary are checked and sorted out. This is no restriction to the adversary, because he can still send any message sequence $(m_n)_{n\in\mathbb{N}}$ by sending $(m_n, num_n)_{n\in\mathbb{N}}$ for an ascending sequence $(num_n)_{n\in\mathbb{N}}$ of natural numbers ($num_n = n$ in the perfect ordered system) until the message bound has been reached.

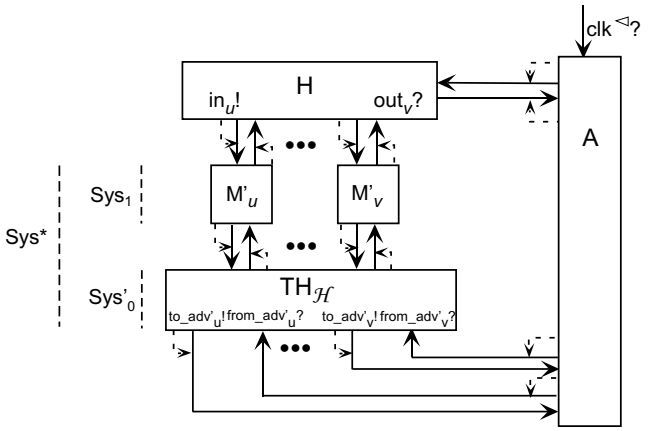

Figure 4.2: The Split Ideal System.

If we now combine the two systems Sys'_0 and Sys_1 in the "canonical" way, i.e., we combine those structure with the same index $\mathcal{H}$ we obtain the system Sys^* that we will refer to as split ideal system (see Figure 4.2). Finally, we define the channels $\mathsf{out}'_u!$ and $\mathsf{in}'_u!$ of Sys^* to be secure.

4.1.3 The Real System

Our real system $Sys^{\#}$ is derived be replacing the ideal specification Sys'_0 with the concrete implementation Sys_0. Similar to the previous combination, we again rename the ports $\mathsf{in}_u?$, $\mathsf{out}_u!$, and $\mathsf{out}_u{}^{\triangleleft}!$ of each structure of Sys_0 into $\mathsf{in}'_u?$, $\mathsf{out}'_u!$, and $\mathsf{out}'_u{}^{\triangleleft}!$, respectively, for every $u \in \mathcal{M}$. Now, we build the combination of Sys_1 and Sys_0 in the canonical way. We obtain a new system $Sys^{\#}$ that we refer to as real ordered system. It is shown in Figure 4.3. Note that the derived system $Sys^{\#}$ is in fact a concrete implementation without containing any abstraction, since both systems Sys_0 and Sys_1 are completely real. Moreover, the system $Sys^{\#}$ can be seen as a standard cryptographic system (cf. Definition 2.17) if we combine the machine M_u and M'_u for every $u \in \mathcal{M}$ and consider this combined machine as the machine of user u.

4.2 Proving Security of the Real Ordered System

After introducing the different schemes, we now focus on proving the security of the real ordered system. We will show that the real ordered system is at least as secure as the specification. This is captured by the following theorem.

Theorem 4.1 *(Security of Real Ordered Secure Message Transmission)* For all $n \in \mathbb{N}$ and $s_1, s_2, L \in \mathbb{N}[x]$, $Sys^{\#} \geq_{\mathsf{sec}}^{f,\mathsf{poly}} Sys^{\mathsf{spec}}$ holds for the canonical mapping f, provided the signature and encryption schemes are secure. This holds with blackbox simulatability. □

Our proof contains the already described four steps, illustrated in Figure 2.3 on page 20. As our first step, we use the result of [49] which yields the relation $Sys_0 \geq_{\mathsf{sec}}^{f_0} Sys'_0$. Secondly, we use the composition theorem (Theorem 2.1), which yields the relation

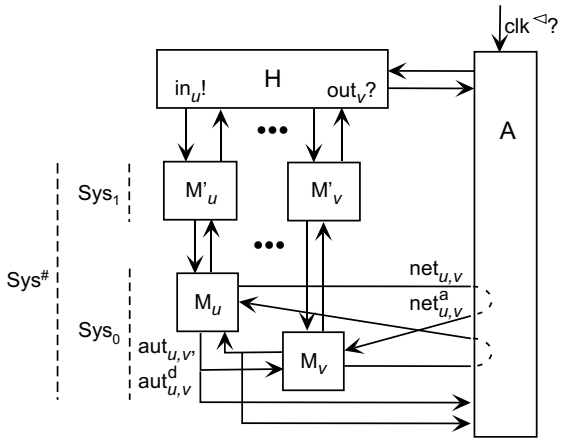

Figure 4.3: Sketch of the Real Ordered System for Secure Message Transmission. Clock-out ports are omitted for the sake of readability.

$Sys^{\#} \geq_{\mathsf{sec}}^{f^{\#}} Sys^{*}$. The only remaining task is to check that its preconditions are in fact fulfilled, which is straightforward since the system Sys_1 is polynomial-time as shown above. Assume that we have already proven $Sys^{*} \geq_{\mathsf{sec}}^{f_1} Sys^{\mathsf{spec}}$, then $Sys^{\#} \geq_{\mathsf{sec}}^{f} Sys^{\mathsf{spec}}$ follows from the transitivity lemma (Lemma 2.6).

Thus, we only have to prove $Sys^{*} \geq_{\mathsf{sec}}^{f_1,\mathsf{poly}} Sys^{\mathsf{spec}}$, but we will even prove the perfect case $Sys^{*} \geq_{\mathsf{sec}}^{f_1,\mathsf{perf}} Sys^{\mathsf{spec}}$.

Lemma 4.1 *For all $n \in \mathbb{N}$ and $s_1, s_2, L \in \mathbb{N}[x]$, $Sys^{*} \geq_{\mathsf{sec}}^{f,\mathsf{perf}} Sys^{\mathsf{spec}}$ holds for the canonical mapping f_1, provided the signature and encryption schemes are secure. This holds with blackbox simulatability.* □

In order to prove this, we assume a configuration $conf_{\mathsf{so}} := (\{\mathsf{TH}_{\mathcal{H}}\} \cup \hat{M}'_u, S_{\mathcal{H}}, \mathsf{H}, \mathsf{A})$ with $\hat{M}'_u = \{\mathsf{M}'_u \mid u \in \mathcal{H}\}$ to be given which we call split-ordered configuration. We then have to show that there exists a configuration $conf_{\mathsf{sp}} := (\{\mathsf{TH}'_{\mathcal{H}}\}, S_{\mathcal{H}}, \mathsf{H}, \mathsf{A}')$, called specification configuration, yielding indistinguishable views for the honest user.

The adversary A' will be constructed by two machines: a so-called simulator $\mathsf{Sim}_{\mathcal{H}}$, that we will define in the following, and the original adversary A, which corresponds to the notion of blackbox simulatability (cf. Definition 2.15). These configurations are shown in Figure 4.4.

We will show that the runs of these configurations can be mapped together in a way that does not change the view of an honest user H. Therefore, we define a relation ϕ on the states of both systems, and we will show that the views of A and H are equal in both configurations provided that the relation ϕ holds during the whole run. We start with the definition of the simulator $\mathsf{Sim}_{\mathcal{H}}$; the definition of the relation ϕ is postponed to the next section. After that, we prove its correctness using formal proof tools.

Definition of the Simulator $\mathsf{Sim}_{\mathcal{H}}$. The Simulator $\mathsf{Sim}_{\mathcal{H}}$ will be inserted between the trusted host $\mathsf{TH}'_{\mathcal{H}}$ and the adversary, see Figure 4.4.

The ports of $\mathsf{Sim}_{\mathcal{H}}$ are given by $\{\mathsf{to\_adv}_u?, \mathsf{from\_adv}_u!, \mathsf{from\_adv}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{\mathsf{from\_adv}'_u?, \mathsf{to\_adv}'_u!, \mathsf{to\_adv}'_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$. The first set contains the ports connected to $\mathsf{TH}'_{\mathcal{H}}$, the ports of the second set are for communication with the adversary. Internally, $\mathsf{Sim}_{\mathcal{H}}$ maintains two arrays $(init^{\mathsf{sim}}_{u,v})_{u,v\in\mathcal{M}}$, $(stopped^{\mathsf{sim}}_{u})_{u\in\mathcal{H}}$ over $\{0,1\}$, an

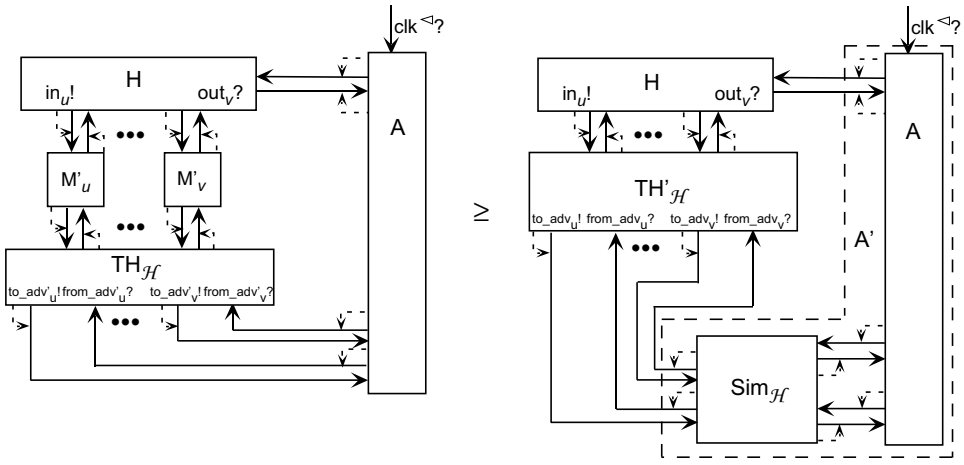

Figure 4.4: Proof Overview of $Sys^{*} \geq^{f,\mathsf{perf}}_{\mathsf{sec}} Sys^{\mathsf{spec}}$.

array $(msg\_out^{\mathsf{sim}}_{u,v})_{u\in\mathcal{A},v\in\mathcal{H}}$ over $\{0,\ldots,s_1(k)\}$, and an array $(sc\_out^{\mathsf{sim}}_{u,v})_{u\in\mathcal{M},v\in\mathcal{H}}$ over $\{0,\ldots,s_2(k)\}$. All four arrays are initialized with 0 everywhere. The arrays match the arrays in the split ideal system, however the array $msg\_out^{\mathsf{sim}}_{u,v}$ will correspond to $msg\_out^{\mathsf{id}}_{u,v}$ of M'_v for dishonest v only.[2] As length bounds for incoming messages $\mathsf{Sim}_{\mathcal{H}}$ uses the same length bounds as the original adversary A for both communication with the system and with A itself. This ensures that in the polynomial case, inputs of $\mathsf{Sim}_{\mathcal{H}}$ will always be of polynomial length. We now define the behaviour of the simulator. In most cases $\mathsf{Sim}_{\mathcal{H}}$ simply forwards inputs to their corresponding outputs, modifying some internal values.

Initialization.

- **Send initialization:** Upon input $(\mathsf{snd\_init})$ at $\mathsf{to\_adv}_u?$, $\mathsf{Sim}_{\mathcal{H}}$ sets $init^{\mathsf{sim}}_{u,u} := 1$ and outputs $(\mathsf{snd\_init})$ at port $\mathsf{to\_adv}'_u!$, 1 at $\mathsf{to\_adv}'^{\triangleleft}_u!$.
- **Receive initialization:** Upon input $(\mathsf{rec\_init}, u)$ at $\mathsf{from\_adv}'_v?$: If $stopped^{\mathsf{sim}}_u = 0$ and $init^{\mathsf{sim}}_{u,v} = 0$ and $[u \in \mathcal{H} \implies init^{\mathsf{sim}}_{u,u} = 1]$ $\mathsf{Sim}_{\mathcal{H}}$ sets $init^{\mathsf{sim}}_{u,v} := 1$. If additionally $sc\_out^{\mathsf{sim}}_{u,v} < s_2(k)$ holds, it sets $sc\_out^{\mathsf{sim}}_{u,v} := sc\_out^{\mathsf{sim}}_{u,v} + 1$ and outputs $(\mathsf{rec\_init}, u)$ at port $\mathsf{from\_adv}_v!$, 1 at $\mathsf{from\_adv}^{\triangleleft}_v!$.

Sending and receiving messages.

- **Send:** Upon input $(\mathsf{send\_blindy}, i, l', v)$ at $\mathsf{to\_adv}_u?$, $\mathsf{Sim}_{\mathcal{H}}$ determines $l := l' + c(k)$ and outputs $(\mathsf{send\_blindy}, i, l, v)$ at port $\mathsf{to\_adv}'_u!$.

  Upon input (send, m, v) at $\mathsf{to\_adv}_u?$, $\mathsf{Sim}_{\mathcal{H}}$ simply forwards the input to port $\mathsf{to\_adv}'_u!$.
- **Receive from honest party** u**:** Upon input $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}'_v?$, $\mathsf{Sim}_{\mathcal{H}}$ forward this input to port $\mathsf{from\_adv}_v!$.

[2] Note that even messages sent by dishonest users are probably sorted out in our split ideal system, but they are simply forwarded in our specifi cation, so these messages have to be sorted out by $\mathsf{Sim}_{\mathcal{H}}$.

- **Receive from dishonest party u:** Upon input $(\mathsf{receive}, u, m')$ at $\mathsf{from\_adv}'_v?$ with $u \in \mathcal{A}$, $\mathsf{Sim}_{\mathcal{H}}$ decomposes $m' = (m, num)$: If $stopped_v^{\mathsf{sim}} = 0$, $init_{v,v}^{\mathsf{sim}} = 1$, $init_{u,v}^{\mathsf{sim}} = 1$, $len(m') < L_1(k)$, $msg\_out_{u,v}^{\mathsf{sim}} \leq num$ ($msg\_out_{u,v}^{\mathsf{sim}} = num$ in the perfect ordered system) and $sc\_out_{u,v}^{\mathsf{sim}} < s_2(k)$, set $msg\_out_{u,v}^{\mathsf{sim}} = num + 1$, $sc\_out_{u,v}^{\mathsf{sim}} := sc\_out_{u,v}^{\mathsf{sim}} + 1$ and output $(\mathsf{receive}, u, m)$ at $\mathsf{from\_adv}_v!$, 1 at $\mathsf{from\_adv}_v{}^{\triangleleft}!$.

- **Stop:** On input (stop) at $\mathsf{from\_adv}'_u?$: If $stopped_u^{\mathsf{sim}} = 0$, $\mathsf{Sim}_{\mathcal{H}}$ sets $stopped_u^{\mathsf{sim}} = 1$ and outputs (stop) at port $\mathsf{from\_adv}_u!$, 1 at $\mathsf{from\_adv}_u{}^{\triangleleft}!$.

What the simulator actually does is recalculating the length of message m into $\mathsf{len}((m, num))$ to achieve indistinguishability. Furthermore, it decomposes messages sent by the adversary, maybe sorting them out, in order to achieve identical outputs in both systems. Now, the overall adversary A', cf. Figure 4.4, is defined by combining A and $\mathsf{Sim}_{\mathcal{H}}$.

Obviously, $\mathsf{Sim}_{\mathcal{H}}$ only makes a polynomial number of steps at each transition. Moreover, $\mathsf{Sim}_{\mathcal{H}}$ will always be scheduled either immediately before or immediately after A (if it is scheduled by the system it either schedules A by construction, or it does not output anything, so A will be scheduled as the master scheduler of the configuration). Thus, the overall adversary A' is polynomial-time if the original adversary is polynomial-time, since A' enters final state if A does, cf. Definition 2.14.

Now the ultimate goal is to show that both collections $\hat{M}_* := \{\mathsf{TH}_{\mathcal{H}}\} \cup \{\hat{M}_u \mid u \in \mathcal{H}\}$ and $\hat{M}_{\mathsf{spec}} := \{\mathsf{TH}'_{\mathcal{H}}, \mathsf{Sim}_{\mathcal{H}}\}$ have the same input-output behaviour, i.e., if they obtain the same inputs they have to produce the same outputs. We will do so by proving a classical deterministic bisimulation, i.e., we will show that the already mentioned relation ϕ is maintained during every step of every trace and that the outputs of both systems are always equal. This is exactly the procedure we will perform using the theorem prover PVS.

Definition 4.1 *(Deterministic Bisimulation) Let two arbitrary collections $\hat{M}_1$ and $\hat{M}_2$ of deterministic machines be given with identical sets of free ports, i.e., $\mathsf{free}([\hat{M}_1]) = \mathsf{free}([\hat{M}_2])$. A deterministic bisimulation between these two collections is a binary relation ϕ on the states of $\hat{M}_1$ and $\hat{M}_2$ such that the following holds.*

- *The initial states of $\hat{M}_1$ and $\hat{M}_2$ satisfy the relation ϕ.*

- *The transition functions δ_1 and δ_2 of $\hat{M}_1$ and $\hat{M}_2$ preserve the relation ϕ and produce identical outputs, i.e., let S_1 and S_2 denote two states of $\hat{M}_1$ and $\hat{M}_2$, respectively, with $(S_1, S_2) \in \phi$. Then for an arbitrary assignment $\mathcal{I}$ of the input ports of $\hat{M}_1$ and $\hat{M}_2$ it holds $(S_1', S_2') \in \phi$ with $(S_1', \mathcal{O}) := \delta_1(S_1, \mathcal{I})$ and $(S_2', \mathcal{O}) := \delta_1(S_2, \mathcal{I})$.*

We call the two collections $\hat{M}_1$ and $\hat{M}_2$ bisimilar, if they are contained in a bisimulation. ◇

Now it is quite easy to see that a deterministic bisimulation between the two collections $\hat{M}_*$ and $\hat{M}_{\mathsf{spec}}$ implies perfect indistinguishability of the view of H, cf. Figure 4.4. Assume for contradiction that the views of H are not identical. Thus, there exists a first time, where either the view of H or the view of the original adversary A (which can tell H this difference afterwards in this case) can be distinguished. Without loss of generality, we assume that this difference occurs at the view of H. However, this difference has to be produced by the system since the adversary A has still identical

views by assumption. Since we claimed it to be the first different step the prior input of both systems had to be identical so both systems produce identical outputs because they are contained in a deterministic bisimulation. Thus, the user obtains identical inputs. Identical inputs cannot be distinguished yielding the desired contradiction.

The next section describes how the actual machines are expressed in the formal syntax of PVS and partly sketches the bisimulation proof.

It is worth mentioning that we used standard paper-and-pencil proofs before we decided to use a formal proof system to validate the desired bisimulation. However, these proofs have turned out to be very error-prone since they are straightforward on the one hand, but long and tedious on the other. In order to illustrate this, we now present a brief excerpt, the "Send initialization" transition, of our original proofs (for our still wrong machines):

Send initialization.

Upon input (snd_init) at $\mathsf{in}_u?$: M'_u checks $stopped_u^{\mathsf{id}} = 0$ and $sc\_in_{u,v}^{\mathsf{id}} < s_1(k)$ for every $v \in \mathcal{M}$. If all these checks are successful, $sc\_in_{u,v}^{\mathsf{id}}$ is increased for every $v \in \mathcal{M}$. Finally it outputs (snd_init) at $\mathsf{in}'_u!$ scheduling it immediately by sending 1 at ${\mathsf{in}'_u}^{\triangleleft}!$. $\mathsf{TH}_{\mathcal{H}}$ then checks $stopped_u^* = 0$ and $init_{u,u}^* = 0$. In case of a previous initialization or stopping signal, it does nothing. So far $\mathsf{TH}'_{\mathcal{H}}$ obviously acts in the same way. $\mathsf{TH}'_{\mathcal{H}}$ checks $sc\_in_{u,v}^{\mathsf{spec}} < s_1(k)$ for every $v \in \mathcal{M}$, increasing $sc\_in_{u,v}^{\mathsf{spec}}$ for every $v \in \mathcal{M}$. The mapping of $sc\_in_{u,v}^{\mathsf{id}}$ to $sc\_in_{u,v}^{\mathsf{spec}}$ ensures identical behaviours. After that it checks $stopped_u^{\mathsf{spec}}$ and $init_{u,u}^{\mathsf{spec}} = 0$ doing nothing if at least one check fails. The mapping of $stopped_u^*$ to $stopped_u^{\mathsf{spec}}$ and $init_{u,u}^*$ to $init_{u,u}^{\mathsf{spec}}$ ensures identical results again.

Now assume $init_{u,u}^* = 0$ (which implies $init_{u,u}^{\mathsf{spec}} = 0$). In $conf_{\mathsf{so}}$, after forwarding (snd_init) by machine M'_u, $\mathsf{TH}_{\mathcal{H}}$ sets $init_{u,u}^* = 1$ and outputs (snd_init) at $\mathsf{to\_adv}'_u!$, 1 at ${\mathsf{to\_adv}'_u}^{\triangleleft}!$. In $conf_{\mathsf{sp}}$, $\mathsf{TH}'_{\mathcal{H}}$ sets $init_{u,u}^{\mathsf{spec}} = 1$ and outputs (rec_init) at $\mathsf{to\_adv}_u!$, 1 at ${\mathsf{to\_adv}_u}^{\triangleleft}!$. The simulator $\mathsf{Sim}_{\mathcal{H}}$ obtains this input at $\mathsf{to\_adv}_u?$ and forwards it to $\mathsf{to\_adv}'_u!$. Obviously, $\phi(init_{u,u}^*) = init_{u,u}^{\mathsf{spec}}$ and $\phi(sc\_in_{u,v}^{\mathsf{id}}) = sc\_in_{u,v}^{\mathsf{spec}}$ remain satisfied and outputs of both systems are identical, so we obtain indistinguishable views with respect to the environment.

Obviously, this paper-and-pencil approach is mainly vulnerable to slow-down of concentration because of its straightforward but tedious manner. However, this transition is, beside the "Stop" transition, by far the easiest case to prove. In particular, the transition of sending and receiving messages turns out to be very large and more difficult to prove, so it is quite probable that parts of the proof are still missing or even wrong. During our formal verification, we in fact found several errors in both our machines and our proofs, which were quite obvious afterwards, but had not been found before. We decided to put the whole paper-and-pencil proof in the web[3], so the reader can make up his own mind.

4.3 Formal Verification of the Bisimulation

4.3.1 Defining the Machines in PVS

In this section, we describe how the still to be proven Lemma 4.1 is formally verified in the theorem proving system PVS [44]. As we already showed in the previous section, it

is sufficient to prove that the two considered collections $\hat{M}_*$ and $\hat{M}_{\mathsf{spec}}$ are contained in a deterministic bisimulation. In order to do so, we first describe how the machines are formalized in PVS. Since the formal machine descriptions are too large to be given here completely, we use the machine $\mathsf{TH}'_{\mathcal{H}}$ as an example how to formalize the machines in PVS. The complete machine descriptions and the proof are available online[3].

We denote the number of participating machines by N, and for a given subset $\mathcal{H} \in \mathcal{ACC}$, we denote the number of honest users by $M := \#\mathcal{H}$. As defined in Scheme 4.1, the machine $\mathsf{TH}'_{\mathcal{H}}$ has $2M$ input ports $\{\mathsf{in}_u? \mid u \in \mathcal{H}\} \cup \{\mathsf{from\_adv}_u? \mid u \in \mathcal{H}\}$. In PVS, we numerate these input ports $1, \ldots, 2M$, where we identify $1, \ldots, M$ with the user ports and $M+1, \ldots, 2M$ with the adversary ports. Similarly, $\mathsf{TH}'_{\mathcal{H}}$ has output ports $\{\mathsf{out}_u! \mid u \in \mathcal{H}\} \cup \{\mathsf{to\_adv}_u! \mid u \in \mathcal{H}\}$, which also are numerated $1, \ldots, 2M$. In PVS, we define the following types to denote machines, honest users, and ports:

```
MACH:     TYPE = subrange(1,N)      %% machines
USERS:    TYPE = subrange(1,M)      %% honest users
PORTS:    TYPE = subrange(1,2*M)    %% port numbers
```

The `subrange(i,j)` type is a PVS built-in type denoting the integers $i, \ldots, j$. We further define an uninterpreted type `STRING` to represent messages.

In Scheme 4.1, the different possible inputs to machine $\mathsf{TH}'_{\mathcal{H}}$ are listed, e.g., $(\mathsf{snd\_init}), (\mathsf{rec\_init}, u), \ldots$ In PVS, the type of input ports is defined using a PVS abstract datatype [43]. The prefix `m1i` in the following stands for "inputs of machine 1" and is used to distinguish between inputs and outputs of the different machines.

```
m1_in_port: DATATYPE
BEGIN
  m1i_snd_init:                                  m1i_snd_init?
  m1i_rec_init(u: MACH):                         m1i_rec_init?
  m1i_send(m: STRING, v: MACH):                  m1i_send?
  m1i_receive_blindly(u: USERS, i: posnat): m1i_receive_blindly?
  m1i_receive(u: MACH, m: STRING):               m1i_receive?
  m1i_stop:                                      m1i_stop?
END m1_in_port
```

This defines an abstract datatype with *constructors* `m1i_rec_init`, `m1i_snd_init` etc. For example, for given u, i, `m1i_receive_blindly(u,i)` constructs an instance of the above datatype, which we identify with $(\mathsf{receive\_blindly}, u, i)$. Given an instance p of this datatype, we can use the *recognizers* on the right side of the definition to distinguish between the different forms. For example, `m1i_receive_blindly?(p)` checks whether the instance p of the `m1i_in_port` datatype was constructed from the `m1i_receive_blindly` constructor. If p was constructed from `m1i_receive_blindly(u,i)` with given u, i, the components u and i can be restored using the *accessor functions* $u(\cdot)$ and $i(\cdot)$; for example, $u(p)$ returns the u component of p. The accessor functions may be overloaded for different constructors (e.g., u is overloaded in our example, namely in `m1i_rec_init`, `m1i_receive_blindly` and `m1i_receive`).

The machine $\mathsf{TH}'_{\mathcal{H}}$ performs a step iff exactly one of the input ports is active (cf. Lemma 2.2). In this case, we call the input *ok*, otherwise *garbage*. The type of the complete inputs to $\mathsf{TH}'_{\mathcal{H}}$ comprising all $2M$ input ports is therefore either garbage, or the number u of the active port together with the input p on port u. This is formalized in the following PVS datatype:

[3] http://www-krypt.cs.uni-sb.de/∼mbackes/PVS/FME2002/

```
M1_INP: DATATYPE
BEGIN
  m1i_garbage:                                  m1i_garbage?
  m1i_ok(u: PORTS, p: m1_in_port):              m1i_ok?
END M1_INP
```

Similar datatypes `m1_out_port` and `M1_OUT` are defined to denote the type of individual outputs, and the type of the complete output of $\mathsf{TH}'_{\mathcal{H}}$, respectively. We omit the details.

Having defined the input and output types of machine $\mathsf{TH}'_{\mathcal{H}}$, it remains to define the state type. As defined in Scheme 4.1, the state of $\mathsf{TH}'_{\mathcal{H}}$ consists of seven one- or two-dimensional arrays. In PVS, arrays are modeled as functions mapping the indices to the contents of the array. For example `[ MACH,USERS -> nat ]` defines a two-dimensional array of natural numbers, where the first index ranges over $\mathcal{M}$, and the second ranges over $\mathcal{H}$. The state type of $\mathsf{TH}'_{\mathcal{H}}$ is defined as a record of such arrays. There is only one small exception: the array $deliver^{\mathsf{spec}}_{u,v}$ stores lists of tuples (m, i) (e.g., see the "Send" transition), where m is a string and $i \in \mathbb{N}$. It is convenient in PVS to decompose this array of lists of tuples into two arrays of lists, where the first array $deliver^{\mathsf{spec}}_{u,v}$ stores lists of messages m, and the second array $deliv\_i^{\mathsf{spec}}_{u,v}$ stores lists of naturals i. Altogether, this yields a state type of eight arrays:

```
M1_STATE: TYPE = [# init_spec: [MACH,MACH -> bool],
                   sc_in_spec: [USERS,MACH -> nat],
                   msg_in_spec: [USERS,MACH -> nat],
                   msg_out_spec: [USERS,USERS -> posnat],
                   sc_out_spec: [MACH,USERS -> nat],
                   deliver_spec: [USERS,USERS -> list[STRING]],
                   deliv_i_spec: [USERS,USERS -> list[posnat]],
                   stopped_spec: [USERS -> bool] #]
```

The initial state `m1_init` is defined as a constant of type `M1_STATE`:

```
M1_init: M1_STATE = (#
  init_spec := LAMBDA (w1,w2: MACH): FALSE,
  ...
  deliv_i_spec := LAMBDA (u1,u2: USERS): null,
  stopped_spec := LAMBDA (u1: USERS): FALSE #)
```

The constructor `null` denotes the empty list. In the definition of machine $\mathsf{TH}'_{\mathcal{H}}$, $sc\_in^{\mathsf{spec}}_{u,v}$ is incremented for all machines v during the "Send initialization" part. This is encapsulated in the following PVS function:

```
incr_sc_in_spec(S: M1_STATE, u: USERS): M1_STATE =
     S WITH [ 'sc_in_spec := LAMBDA (w: USERS, v: MACH):
                  IF w=u THEN S'sc_in_spec(w,v)+1 ELSE
                              S'sc_in_spec(w,v) ENDIF ];
```

The `WITH` construct leaves the record S unchanged except for the `sc_in_spec` component, which is replaced by the λ-expression. The machine $\mathsf{TH}'_{\mathcal{H}}$ is now formalized in PVS as a next-state/output function mapping current state and inputs to the next state and outputs. We exemplarily give the first few lines of the PVS code:

```
M1_ns(S: M1_STATE, I: M1_INP): [# ns: M1_STATE, O: M1_OUT #] =
  IF m1i_garbage?(I) THEN
     (# ns:=S, O:=m1o_garbage #)
        %% do not change the state, output garbage
```

```
ELSE
  LET ua1=ua(I), p=p(I) IN
    %% ua1 is the active port number,
    %% p is the input on this port
  IF ua1<=M AND m1i_snd_init?(p) THEN
    %% we have a send-init on a user port (<=M);
    IF (FORALL w1: S‘sc_in_spec(ua1,w1)<s1k) THEN
      IF S‘init_spec(ua1,ua1) OR S‘stopped_spec(ua1) THEN
        (# ns:=incr_sc_in_spec(S,ua1),O:=m1o_garbage #)
        %% increment sc_in_spec, but do not send any output
      ELSE
        (# ns:=incr_sc_in_spec(S,ua1)
          WITH [ ‘init_spec(ua1,ua1) := TRUE ],
        O := m1o_ok(M+ua1, m1o_snd_init) #)
        %% increment sc_in_spec, set init_spec(ua1,ua1):=true
        %% send m1o_snd_init to adversary port M+ua1
      ENDIF
    ELSE %% otherwise do nothing
      (# ns:=S, O:=m1o_garbage #)
    ENDIF
  ELSIF ua1>M AND m1i_rec_init?(p) THEN
  ...
```

In a similar way we have formalized the machines $\mathsf{TH}_{\mathcal{H}}$, $\{\mathsf{M}'_u \mid u \in \mathcal{H}\}$, and $\mathsf{Sim}_{\mathcal{H}}$. The M machines M'_u in the left part of Figure 4.4 have been combined to a single machine in PVS; however, this is only syntactic and does not change the semantics. The combination of the machines $\mathsf{TH}_{\mathcal{H}}$ and $\{\mathsf{M}'_u \mid u \in \mathcal{H}\}$ respectively $\mathsf{TH}'_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ is straightforward by composition of the corresponding state transition functions.

The only non-trivial choice we have made in the transliteration of the machines to PVS is the type of the input- and output-ports. In a previous attempt, we did not use the abstract datatype definition of `M1_INP`, but defined `M1_INP` as an array of $2M$ individual input ports; in order to model non-active ports, we added an `m1i_inactive` form to the input port type `m1i_in_port`. An input from `M1_INP` was defined to be *ok* iff exactly one of the ports is different from `m1i_inactive`. This obviously models the same valid inputs as the definition of `M1_INP` above. The problem with the array definition is that extracting the active port number u involves an application of the choice-function ε in order to choose the index u of the array for which the port is active. The application of the choice-function considerably complicates the proofs in PVS, since the definition of ε is not constructive in PVS. In contrast, in the definition using the abstract datatype, the active port number u can be constructively extracted from the input by applying the accessor function of the abstract datatype. Due to constructiveness, the proofs in PVS become much simpler. This problem in the port definition also applies to the output ports of the machines.

The rest of the transliteration of the machine definitions to PVS is easy and straightforward. In the following, we revert to standard mathematical notation for the sake of brevity and readability. However, it should be noted once more that all the definitions and claims in this section have been formalized and verified in PVS.

4.3.2 Proving the Bisimulation

In order to prove Lemma 4.1, we prove the following predicates to be invariants of the considered collections $\hat{M}_*$ and $\hat{M}_{\mathsf{spec}}$ during every trace of every configuration.

- $stopped^* = stopped^{\mathsf{id}} = stopped^{\mathsf{sim}} = stopped^{\mathsf{spec}}$.
  Note that we compare whole arrays in this predicate, i.e., we make use of the higher-order capabilities of PVS. One could also write $\forall u : stopped^*_u = stopped^{\mathsf{id}}_u = \ldots$, but the equality of the whole arrays is more concise and easier to use in the proofs.
- $sc\_in^{\mathsf{id}} = sc\_in^{\mathsf{spec}}$.
- $init^* = init^{\mathsf{sim}} = init^{\mathsf{spec}}$.
- $msg\_in^{\mathsf{id}} = msg\_in^{\mathsf{spec}}$.
- $\forall u, v \in \mathcal{H} : length(deliver^*_{u,v}) = length(deliv\_i^*_{u,v})$.
  $length$ is the PVS function delivering the length of lists. Note that we use the quantified form of the invariant here instead of the higher-order form, since otherwise we would have to 'lift' the $length$ function to arrays of lists.
- $\forall u, v \in \mathcal{H} : length(deliver^{\mathsf{spec}}_{u,v}) = length(deliv\_i^{\mathsf{spec}}_{u,v})$.
- $deliver^* = deliver^{\mathsf{spec}}$ and $deliv\_i^* = deliv\_i^{\mathsf{spec}}$.
- $sc\_out^{\mathsf{id}} = sc\_out^{\mathsf{spec}}$.
- $\forall w \in \mathcal{M}, u \in \mathcal{H} : sc\_out^{\mathsf{sim}}_{w,u} \leq sc\_out^{\mathsf{spec}}_{w,u}$.
  Again we use the quantified form, since otherwise we had to lift "$\leq$" to arrays.
- $\forall w \in \mathcal{M}, u \in \mathcal{H} : (w \in \mathcal{H} \implies msg\_out^{\mathsf{id}}_{w,u} = msg\_out^{\mathsf{spec}}_{w,u})$ and
  $(w \in \mathcal{A} \wedge sc\_out^{\mathsf{id}}_{w,u} < s_2(k) \implies msg\_out^{\mathsf{id}}_{w,u} = msg\_out^{\mathsf{sim}}_{w,u})$

Each of the 10 invariants is formalized as a predicate $\phi_i(S_{\mathsf{si}}, S_{\mathsf{sp}})$ on the current states of the two collections $\hat{M}_*$ and $\hat{M}_{\mathsf{spec}}$. The conjunction of all the ϕ_i yields the already mentioned relation ϕ. Let δ_{si} and δ_{sp} denote the overall transition function of the considered machine collections $\hat{M}_*$ and $\hat{M}_{\mathsf{spec}}$, respectively. The following theorem asserts, that the invariants indeed are invariants of the systems:

Theorem 4.2 *Let S_{si} and S_{sp} be states of the two collections $\hat{M}_*$ and $\hat{M}_{\mathsf{spec}}$ such that all invariants $\phi_i(S_{\mathsf{si}}, S_{\mathsf{sp}})$, $1 \leq i \leq 10$ hold. The transition functions $\delta_{\mathsf{si}}, \delta_{\mathsf{sp}}$ preserve the invariants, i.e., for an arbitrary assignment $\mathcal{I}$ of the inputs ports of $\hat{M}_*$ and $\hat{M}_{\mathsf{spec}}$ it holds*

$$\phi_i(S'_{\mathsf{si}}, S'_{\mathsf{sp}})\ \forall i, 1 \leq i \leq 10$$

with $(S'_{\mathsf{si}}, \mathcal{O}_{\mathsf{si}}) := \delta_{\mathsf{si}}(S_{\mathsf{si}}, \mathcal{I})$ and $(S'_{\mathsf{sp}}, \mathcal{O}_{\mathsf{sp}}) := \delta_{\mathsf{sp}}(S_{\mathsf{sp}}, \mathcal{I})$. Furthermore, the initial states $initial_{\mathsf{si}}$ and $initial_{\mathsf{sp}}$ satisfy all 10 invariants. □

In PVS, this theorem is split into 10 lemmas, one for each invariant. Using the invariants ϕ_i, we prove the following theorem:

Theorem 4.3 *Let S_{si} and S_{sp} be states satisfying all invariants $\phi_i(S_{\mathsf{si}}, S_{\mathsf{sp}})$, $1 \leq i \leq 10$, and let $\mathcal{I}$ be an assignment of the input ports (connected to the users and the adversary) of the collections $\hat{M}_*$ and $\hat{M}_{\mathsf{spec}}$. Then both collections have the same outputs on all ports to the users and the adversary.* □

Together, Theorems 4.2 and 4.3 prove that the two systems are bisimilar, which finishes our proof of Theorem 4.1.

4.3.3 Verification Effort

The manual proof effort in PVS is rather small. The proofs make heavy use of the built-in PVS strategy `(grind)`, which expands definitions and performs automatic case-splitting. The main effort was to figure out the correct parameters for the `(grind)` command. The proof goals not resolved by `(grind)` were proved with little manual assistance.

However, looking for errors and thinking about the necessary modifications of the machines was a time-consuming task which took the bulk of our time. During our proof attempts, we simultaneously debugged the machines until we finally found the correct specifications of all machines. After that, the proof itself turned out to be quite easy. Putting it all together, the formalization of the machines in PVS took 2 weeks, and the development of the proofs took another week. A complete checking of the proof takes about one hour on a 600 MHz Athlon processor.

4.4 Summary

We have presented the first abstract specification for secure message transmission preventing message reordering. Moreover, we derived a secure, concrete implementation. Its proof of security is based on the composition theorem 2.1, the proof of [49], and on a bisimulation which we formally verified using the theorem prover PVS. Our approach furthermore comprises a general strategy how to derive concrete implementations by splitting specifications into smaller systems that can then be refined stepwise using the composition theorem and formal proof systems, which yields trustworthy proofs. Moreover, these implementations are secure with respect to the cryptographic definitions, so we have finished the first step in combining the advantages of formal proof systems and cryptography.

If we look back at Figure 1.1 in the introduction, we formally proved the upper-left arrow, i.e., we showed that our specification uses the abstract primitive for secure message transmission along with a filtering system Sys_1. The remaining task is to verify the claimed integrity property, i.e., prevention of message reordering, which corresponds to the upper-right arrow. This task will be the topic of the next chapter.

Chapter 5

Sound Verification of Integrity Properties

In this chapter, we consider tool-assisted verification of integrity properties of cryptographic protocols. At first, we formally define what integrity properties in fact are in our model, and what it means that a system fulfills them. After that, we show that integrity properties are preserved under step-wise refinement, so formal verifications of our abstractions carry over to the concrete counterparts which provides the up to now missing link between between formal proof tools and cryptography. Moreover, we show that logic derivations among integrity properties in asynchronous scenarios are valid for the concrete systems in the cryptographic sense, which makes them accessible to theorem provers.

We conclude with a formal validation of our specification of the previous section, i.e., we show that message reordering is in fact prevented. We again use PVS to obtain a formally verified proof. According to our results, the proof automatically carries over to the concrete implementation which yields the first machine-aided, but nevertheless sound proof of the concrete goals of a concrete system. Together with the result of the previous chapter, this completes the first tool-supported and cryptographically sound proof of both the security of a concrete implementation and its actual integrity goals.

5.1 Integrity Requirements

In this section, we show how the relation "at least as secure as" relates to properties a system should fulfill, e.g., safety requirements expressed in temporal logic. 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 of specified ports, since requirements of various parties in cryptography are often made for different trust assumptions.

Definition 5.1 (Integrity Requirements) *An integrity requirement Req for a system Sys is a function that assigns a set of valid traces at the ports in S to each set S with $(\hat{M}, S) \in Sys$. More precisely such a trace is a sequence $(v_i)_{i\in\mathbb{N}}$ of values over port names and Σ^*, so that v_i is of the form $v_i := \bigcup_{p\in S}\{p : v_{p,i}\}$ and $v_{p,i} \in \Sigma^*$. For the*

computational and statistical case, the trace has to be finite. We say that Sys fulfills Req

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

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

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

The class $SMALL$ must be closed under addition and making functions smaller.

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

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

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

We now prove that integrity properties of abstract systems carry over to their concrete counterparts, i.e., if the properties are valid for the abstract system and as secure as a real system, this real system also fulfills concrete versions of these goals.

5.2 The Preservation Theorem

Theorem 5.1 (Conservation of Integrity Properties) *Let a system Sys_2 be given that fulfills an integrity requirement Req, and let $Sys_1 \geq^{f}_{\mathsf{sec}} Sys_2$ for a valid mapping f, i.e., we only map structures with an identical set of specified ports. Then also $Sys_1 \models Req$. This holds in the perfect and statistical sense, and in the computational sense if membership in the set $Req(S)$ is decidable in polynomial time for all S.* □

Proof. Req is well-defined on Sys_1, since simulatability implies that for each $(\hat{M}_1, S_1) \in Sys_1$ there exists $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ with $S_1 = S_2$. We will now prove that if Sys_1 does not fulfill the requirement, the two systems could be distinguished yielding a contradiction.

Assume that a configuration $conf_1 = (\hat{M}_1, S_1, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1)$ contradicts the theorem. In order to show that this contradiction can be used to distinguish the two systems, we need an honest user that connects to *all* specified ports. Otherwise, the contradiction might stem from those specified ports which are connected to the adversary, but those ports are not considered by simulatability.

However, recall that we already proved in Section 3.2, that standard simulatability is equivalent to s-simulatability. Thus, we can without loss of generality restrict our attention to those configurations where the honest user connect to *all* specified ports of the considered structure. Alternatively, we can apply Lemma 3.3, i.e., there is a configuration $conf_{*,1} = (\hat{M}_1, S_1, \mathsf{H}_\mathsf{s}, \mathsf{A}_\mathsf{s})$ in which the user connects to all specified ports,

with $run_{conf_{*,1}}\lceil_{S_1} = run_{conf_1}\lceil_{S_1}$, so $conf_{*,1}$ also contradicts the theorem. Moreover, $conf_{*,1}$ is polynomial-time if $conf_1$ is polynomial-time.

Note that all specified ports are now connected to the honest user; thus, we can exploit simulatability. Looking at the proof of Lemma 3.3 we see that $conf_{*,1}$ is a suitable configuration if $conf_1$ is suitable. Hence, there exists an indistinguishable configuration $conf_{*,2} = (\hat{M}_2, S_2, \mathsf{H_s}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_2)$, i.e., $view_{conf_{*,1}}(\mathsf{H_s}) \approx view_{conf_{*,2}}(\mathsf{H_s})$. By our assumption, the requirement is fulfilled for this configuration (perfectly, statistically, or computationally). Furthermore, the view of $\mathsf{H_s}$ in both configurations contains the trace at $S := S_1 = S_2$, i.e., the trace is a function $\lceil_S$ of the view.

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 any polynomial l be given. The statistical distance $\Delta(view_{conf_{*,1},k,l(k)}(\mathsf{H_s}),\ view_{conf_{*,2},k,l(k)}(\mathsf{H_s}))$ is a function $g(k) \in SMALL$. We apply Lemma 2.4 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 Dist as follows: Given the view of machine $\mathsf{H_s}$, it extracts the run restricted to S and verifies whether the result 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 $\mathsf{H_s}$ is of polynomial length, and membership in $Req(S)$ was required to be polynomial-time decidable. Its advantage in distinguishing is

$$\begin{aligned} & |P(\mathsf{Dis}(1^k, view_{conf_{*,1},k}) = 1) - P(\mathsf{Dis}(1^k, view_{conf_{*,2},k}) = 1)| \\ = \ & |P(run_{conf_{*,1},k}\lceil_S \notin Req(S)) - P(run_{conf_{*,2},k}\lceil_S \notin Req(S))|. \end{aligned}$$

Since the second term is negligible by assumption, and $NEGL$ is closed under addition, the first term also has to be negligible, yielding the desired contradiction. ■

We now show that if integrity requirements are formulated in a logic, e.g., temporal logic, abstract derivations in the logic are valid in the cryptographic sense.

Theorem 5.2

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 all 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. ∎

For applying this theorem to concrete logics, we have to show that the common deduction rules hold. As an example, we consider modus ponens, i.e., if one has derived that 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

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

From Theorem 5.2b, we conclude $Sys \models Req_a \cap Req_{a \rightarrow b}$. Obviously, $Req_a \cap Req_{a \rightarrow b} = Req_{a \wedge b} \subseteq Req_b$ holds, so the claim follows from Theorem 5.2a.

5.3 Validation of the Ordered Channel Specification

In this section, we formally verify the actual integrity property of our specification of the previous chapter, i.e., we formally prove that message reordering is in fact prevented. Moreover, we already presented a concrete, secure implementation in the previous chapter, so our verification automatically carries over to this concrete system using our preservation theorem of the previous section.

5.3.1 The Integrity Property

As we have already seen above, the considered specification has been designed to fulfill the property that the order of messages is maintained during every trace of the configuration. Thus, our goal is to prove that for arbitrary traces, arbitrary users $u, v \in \mathcal{H}$, $u \neq v$, and any point in time, the messages that v received so far by u via $\mathsf{TH}'_{\mathcal{H}}$ are indeed a sublist of the messages which have been sent by u to $\mathsf{TH}'_{\mathcal{H}}$ aimed for forwarding to v. The former list is called *receive-list*, the latter *send-list*.

At first sight, the property seems to hold by construction, but experience shows that such proofs made by "simply looking" are often flawed with hindsight. However, even if proofs of this kind are made by hand in a rigorous way, they often turn out to be apparently straightforward and dull which yields proofs with faults and imperfections. In order to obtain trustworthy proofs, we again use the theorem proving system PVS along with our machine encoding of $\mathsf{TH}'_{\mathcal{H}}$ of the previous chapter, and formally verify the integrity property. This will be described in the following. For reasons of readability and brevity, we use standard mathematical notation instead of PVS syntax. The PVS sources are available online.[1]

The formalization of the machine $\mathsf{TH}'_{\mathcal{H}}$ in PVS has already been described in detail in the previous chapter. Here, we assume that the machine operates on an input set I, a state set S, and an output set O. The machine is specified by a state transition function $\delta : I \times S \rightarrow S$ and an output function $\omega : I \times S \rightarrow O$.

In order to formulate the property, we need a PVS-suited, formal notation of (infinite) runs of a machine, of lists, of what it means that a list l_1 is a sublist of a list l_2, and we need formalizations of the *receive-list* and *send-list*.

Definition 5.2 *(Input sequence, state-trace, output sequence) Let* M *be a machine with input set* I*, state set* S*, output set* O*, transition function* δ*, and output function* ω*. Call* $s_{init} \in S$ *the initial state. An* input sequence $i : \mathbb{N}_0 \rightarrow I$ *for machine* M *is a function*

[1] http://www-krypt.cs.uni-sb.de/∼mbackes/PVS/CAV2002

mapping the time (modeled as the set $\mathbb{N}_0$) to inputs $i(t) \in I$. A given input sequence i defines a sequence of states $s^i : \mathbb{N}_0 \to S$ of the machine M *by the following recursive construction:*

$$\begin{aligned} s^i(0) &:= s_{init}, \\ s^i(t+1) &:= \delta(i(t), s^i(t)). \end{aligned}$$

The sequence s^i is called state-trace of M under i. *The* output sequence $o^i : \mathbb{N}_0 \to O$ *of the run is defined as*

$$o^i(t) := \omega(i(t), s^i(t)).$$

We omit the index i if the input sequence is clear from the context. For components x of the state type, we write $x(t)$ for the content of x in $s(t)$. For example, we write $deliver^{\mathsf{spec}}_{u,v}(t)$ to denote the content at time t of the list $deliver^{\mathsf{spec}}_{u,v}$, which is part of the state of $\mathsf{TH'}_{\mathcal{H}}$. ◇

Note that these definitions precisely match our model-intern definition of the view of the machine M. In the context of $\mathsf{TH'}_{\mathcal{H}}$, the input sequence i consists of the messages that the honest users and the adversary send to $\mathsf{TH'}_{\mathcal{H}}$.

Definition 5.3 *(Lists) Lists are defined as a recursive algebraic abstract datatype in the PVS library [43]. We restate the definition here for the sake of completeness. A* list over type T *is the closure of applications of the constructor $null$ yielding an empty list, and the constructor $cons(car : T, cdr : list[T])$ yielding a list with head car and tail cdr. It holds $car(cons(t, l)) = t$ and $cdr(cons(t, l)) = l$. The predicates $null?(l)$ and $cons?(l)$ are used to test whether l is empty or non-empty, respectively. PVS provides functions $length(l)$, $append(t, l)$, and $nth(l, i)$ to measure the length of a list l, to append an element t at the end of the list l, and to access the i^{th} element of l (counted from 0).* ◇

Definition 5.4 *(Sublists) A list l_1 is called sublist of a list l_2 (written $l_1 \subseteq l_2$) iff the following recursive predicate is satisfied:*

$$\begin{aligned} l_1 \subseteq l_2 : \Longleftrightarrow \quad & null?(l_1) \vee \\ & cons?(l_1) \wedge \ \big(car(l_1) = car(l_2) \wedge cdr(l_1) \subseteq cdr(l_2) \\ & \qquad \vee\ l_1 \subseteq cdr(l_2)\big). \end{aligned}$$

Let $k \in \mathbb{N}_0$. The list l_1 is called sublist of the k-prefix of l_2 (written $l_1 \subseteq^k l_2$) iff the following recursive predicate is satisfied:

$$\begin{aligned} l_1 \subseteq^k l_2 : \Longleftrightarrow \quad & null?(l_1) \vee \\ & cons?(l_1) \wedge k \geq 1 \wedge \ \big(car(l_1) = car(l_2) \wedge cdr(l_1) \subseteq^{k-1} cdr(l_2) \\ & \qquad \vee\ l_1 \subseteq^{k-1} cdr(l_2)\big). \end{aligned}$$

◇

The following lemma summarizes some facts on lists and sublists:

Lemma 5.1 *Let l_1, l_2, l_3 be lists over some type T, let $t \in T$, and $k, k' \in \mathbb{N}_0$. It holds:*

1. $k \leq length(l_1) \implies$
$$nth(append(t, l_1), k) = \begin{cases} nth(l_1, k) & \text{if } k < length(l_1) \\ t & \text{otherwise} \end{cases}$$

2. $l_1 \subseteq l_2 \implies l_1 \subseteq append(t, l_2)$
3. $l_1 \subseteq l_2 \implies append(t, l_1) \subseteq append(t, l_2)$
4. $l_1 \subseteq^k l_2 \implies l_1 \subseteq^k append(t, l_2)$
5. $k < length(l_2) \wedge l_1 \subseteq^k l_2 \implies append(nth(l_2, k), l_1) \subseteq^{k+1} l_2$,
   that is, one may append the k^{th} element (counted from 0) of l_2 to l_1 while preserving the prefix-sublist property.
6. $k' \geq k \wedge l_1 \subseteq^k l_2 \implies l_1 \subseteq^{k'} l_2$
7. $l_1 \subseteq^k l_2 \implies l_1 \subseteq l_2$
8. $l_1 \subseteq l_2 \wedge l_2 \subseteq l_3 \implies l_1 \subseteq l_3$

□

All claims are proved by induction on the recursive structure of the lists. Exemplarily, we give a detailed transcript of the proof of the 8-th claim of the lemma, i.e., we prove transitivity of sublists.

Proof. (Lemma 5.1.8) We prove the claim by induction over l_3. The induction base is trivial, by expanding the definition of sublists in $l_1 \subseteq l_2$ and $l_2 \subseteq null$ which implies $null \subseteq null$.

Let us now turn to the induction step, i.e., we have $l_3 = cons(h_3, t_3)$. As induction hypothesis, we have

$$l \subseteq l' \text{ and } l' \subseteq t_3 \Rightarrow l \subseteq t_3,$$

for all lists l, l'. Moreover, we know that $l_1 \subseteq l_2$ and $l_2 \subseteq cons(h_3, t_3)$. We have to show that $l_1 \subseteq cons(h_3, t_3)$ holds. We now expand the definition of sublists in $l_1 \subseteq cons(h_3, t_3)$, which yields two cases:

1. $car(l_1) \neq h_3$ or $l_1 = null$. Thus, we have to show that $l_1 \subseteq t_3$, or that $l_1 = null$. We now use our induction hypothesis with $l := l_1$ and $l' := l_2$ which yields

   $$l_1 \subseteq l_2 \wedge l_2 \subseteq t_3 \Rightarrow l_1 \subseteq t_3. \tag{5.1}$$

   Hence, it is sufficient to show $l_2 \subseteq t_3$. We now expand the definition of sublists in $l_2 \subseteq cons(h_3, t_3)$ which yields three cases again:

   (a) $l_2 = null$. We expand the definition of sublists in $l_1 \subseteq l_2$ which immediately yields $l_1 = null$ and we are done.

   (b) $l_2 = cons(h_2, t_2)$ and $h_2 \neq h_3$ and $l_2 \subseteq t_3$. Thus, equation 5.1 yields $l_1 \subseteq t_3$ and we are done.

   (c) $l_2 = cons(h_2, t_2)$ and $h_2 = h_3$ and $t_2 \subseteq t_3$. We apply our induction hypothesis again with $l := l_1$ and $l' := t_2$ which yields

   $$l_1 \subseteq t_2 \wedge t_2 \subseteq t_3 \Rightarrow l_1 \subseteq t_3. \tag{5.2}$$

   We now expand the definition of sublists in $l_1 \subseteq l_2$ which yields $l_1 \subseteq t_2$ (the case $l_1 = null$ is trivial, and $h_1 = h_2$ cannot happen because of $h_1 \neq h_3$ and $h_2 = h_3$). Thus, the claim follows from equation 5.2.

2. $car(l_1) = h_3$ or $l_1 = null$. Thus, we have to show $cdr(l_1) \subseteq t_3$ or $l_1 = null$. We now expand the definition of sublists in $l_2 \subseteq cons(h_3, t_3)$ which yields three cases:

   (a) $l_2 = null$. This case is trivial again by expanding the definition of $l_1 \subseteq l_2$.

   (b) $l_2 = cons(h_2, t_2)$ and $h_2 = h_3$ and $t_2 \subseteq t_3$. We now use our induction hypothesis with $l := cdr(l_1)$ and $l' := t_2$ which yields

   $$cdr(l_1) \subseteq t_2 \wedge t_2 \subseteq t_3 \Rightarrow cdr(l_1) \subseteq t_3.$$

   Thus, it is sufficient to show $cdr(l_1) \subseteq t_2$. We now use a simple lemma about sublists, which we formally proved in PVS, but only state here for the sake of readability:

   $$l_1 \subseteq l_2 \wedge l_1 \neq null \wedge l_2 \neq null \Rightarrow cdr(l_1) \subseteq cdr(l_2).$$

   Because of $l_1 \neq null$ and $l_2 \neq null$, the claim follows.

   (c) $l_2 = cons(h_2, t_2)$ and $h_2 \neq h_3$ and $l_2 \subseteq t_3$. We use our induction hypothesis with $l := l_1$ and $l' := l_2$ which yields

   $$l_1 \subseteq l_2 \wedge l_2 \subseteq t_3 \Rightarrow l_1 \subseteq t_3.$$

   The claim follows.

■

Moreover, we give the full PVS proof of transitivity of sublists in the Appendix, so the reader is encouraged to compare the above transcript with this machine-generated proof.

The seven remaining claims can be proven similarly, so we simply refer to our PVS proofs on our web page again.

Definition 5.5 *(Receive- and send-list) Let i be an input sequence for machine $\mathsf{TH}'_{\mathcal{H}}$, and let s and o be the corresponding state-trace of $\mathsf{TH}'_{\mathcal{H}}$ and the output sequence, respectively. Let $u, v \in \mathcal{H}$. The receive-list is obtained by appending a new element m whenever v receives a message $(\mathsf{receive}, m, u)$ from $\mathsf{TH}'_{\mathcal{H}}$. The send-list is obtained by appending m whenever u sends a message (send, m, v) to $\mathsf{TH}'_{\mathcal{H}}$. Formally, this is captured in the following recursive definitions:*

$$recvlist^i_{u,v}(t) := \begin{cases} null & \text{if } t = -1, \\ append(m, recvlist^i_{u,v}(t-1)) & \text{if } o^i(t) = (\mathsf{receive}, m, u) \\ & \text{at } \mathsf{out}_v! \wedge t \geq 0. \\ recvlist^i_{u,v}(t-1) & \text{otherwise} \end{cases}$$

$$sendlist^i_{u,v}(t) := \begin{cases} null & \text{if } t = -1, \\ append(m, sendlist^i_{u,v}(t-1)) & \text{if } i(t) = (\mathsf{send}, m, v) \\ & \text{at } \mathsf{in}_u? \wedge t \geq 0. \\ sendlist^i_{u,v}(t-1) & \text{otherwise} \end{cases}$$

◇

We are now ready to give a precise, PVS-suited formulation of the integrity property we are aiming to prove:

Theorem 5.3 *For any* $\mathsf{TH}'_{\mathcal{H}}$ *input sequence* i*, for any* $u, v \in \mathcal{H}$*,* $u \neq v$*, and any point in time* $t \in \mathbb{N}$*, it holds*

$$\mathit{recvlist}^{i}_{u,v}(t) \subseteq \mathit{sendlist}^{i}_{u,v}(t). \tag{5.3}$$

□

Proof (sketch). In the following, we omit the index i. The proof is split into two parts: we prove $\mathit{recvlist}_{u,v}(t-1) \subseteq \mathit{deliver}^{\mathsf{spec}}_{u,v}(t)$ and $\mathit{deliver}^{\mathsf{spec}}_{u,v}(t) \subseteq \mathit{sendlist}_{u,v}(t-1)$. The claim of the theorem then follows from Lemma 5.1.8.

The second claim $\mathit{deliver}^{\mathsf{spec}}_{u,v}(t) \subseteq \mathit{sendlist}_{u,v}(t-1)$ is proved by induction on t. Both induction base and step are proved in PVS by the built-in strategy `(grind)`, which performs automatic definition expanding and rewriting with Lemma 5.1.

The first claim $\mathit{recvlist}_{u,v}(t-1) \subseteq \mathit{deliver}^{\mathsf{spec}}_{u,v}(t)$ is more complicated. The claim is also proved by induction on t. However, it is quite easy to see that the claim is not inductive: in case of a $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}_v?$, $\mathsf{TH}'_{\mathcal{H}}$ outputs $(\mathsf{receive}, m, u)$ to $\mathsf{out}_v!$, where $(m, j) := \mathit{deliver}^{\mathsf{spec}}_{u,v}[i]$, i.e., m is the ith message of the $\mathit{deliver}^{\mathsf{spec}}_{u,v}$ list (cf. scheme 4.1 on page 60). By the definition of the receive-list, the message m is appended to $\mathit{recvlist}_{u,v}$. In order to prove that $\mathit{recvlist}_{u,v} \subseteq \mathit{deliver}^{\mathsf{spec}}_{u,v}$ is preserved during this transition, it is necessary to know that the receive list was a sublist of the prefix of the $\mathit{deliver}^{\mathsf{spec}}_{u,v}$ list that does not reach to m. It would suffice to know that

$$\mathit{recvlist}_{u,v}(t-1) \subseteq^{i} \mathit{deliver}^{\mathsf{spec}}_{u,v}(t).$$

Then the claim follows from lemma 5.1.5.

We therefore strengthen the invariant to comprise the prefix-sublist property. However, the i in the above prefix-sublist relation stems from the input $(\mathsf{receive\_blindly}, u, i)$, and hence is not suited to state the invariant. To circumvent this problem, we recursively construct a sequence $\mathit{last\_rcv\_blindly}_{u,v}(t)$ which holds the parameter i of the last valid $(\mathsf{receive\_blindly}, u, i)$ received by $\mathsf{TH}'_{\mathcal{H}}$ on $\mathsf{from\_adv}_v?$; then

$$\mathit{recvlist}_{u,v}(t-1) \subseteq^{l} \mathit{deliver}^{\mathsf{spec}}_{u,v}(t) \text{ with } l = \mathit{last\_rcv\_blindly}_{u,v}(t)$$

is an invariant of the system. We further strengthen this invariant by asserting that $\mathit{last\_rcv\_blindly}_{u,v}(t)$ and the j's stored in the $\mathit{deliver}^{\mathsf{spec}}_{u,v}$ list grow monotonically. Together this yields the inductive invariant. We omit the technical details, and refer the reader to the PVS files available online[1]. ■

Applying our standard definition 5.1 of integrity requirements, we can now define that the requirement Req holds for an arbitrary trace tr if and only if Equation 5.3 holds for all $u, v \in \mathcal{H}$, $u \neq v$, and the input sequence i of the given trace tr. Thus, Theorem 5.3 can be rewritten in standard notation as $[(run_{conf,k}\lceil_S)] \subseteq Req(S)$ for all k, i.e., we have proven that the specification Sys^{spec} perfectly fulfills the integrity requirement $Req(S)$. In formulas, $Sys^{\mathsf{spec}} \models_{\mathsf{perf}} Req(S)$.

5.3.2 Verification Effort

Together, the development of the inductive invariant and its proof took 2 weeks, which included some failed approaches in strengthening the invariant to become inductive. The proof of the invariant takes 500 proof commands, i.e., steps with manual assistance.

A further week and 350 proof commands were needed for the development of the sublist theory, which can be reused in future verification projects.

The main difficulty during the verification of the invariant was finding the stronger inductive invariant. Once the correct invariant was found, its proof was relatively easy. Before we started the formal verification, we had a hand-written proof of theorem 5.3. However, the proof was incomplete in the sense that we did not prove some needed invariants; in fact, we did not even notice that we used these invariants in our hand-made proofs, because of our intuitive understanding of the system.

5.4 Conclusion

In this chapter, we continued and succeeded in developing a general methodology how cryptographic protocols in asynchronous reactive systems can be verified both sound with respect to cryptographic definitions and machine-aided. As the last remaining step we considered an actual integrity goal a protocol should fulfill on the abstract and the concrete level. Moreover, we showed that verification of abstract goals automatically carries over to their concrete counterparts in case of integrity properties, and we showed that logic derivations among integrity properties are valid for the concrete systems in the cryptographic sense, which makes them accessible to theorem provers. As an example, we formally validated our scheme for ordered secure message transmission of the precedent chapter using the theorem proving system PVS [44].

If we consider our approach as a whole, i.e., combining the results of [49], of Chapter 4, and of this chapter, we can state the following results: we successfully finished presenting the first approach for cryptographic protocol verification that is sound with respect to the underlying cryptographic primitives, and that allows abstractions suitable for formal proof systems. These abstractions are easy to use and can be combined to large protocols very easily; moreover these abstractions can be refined step-wise in order to obtain concrete systems, and proofs on the abstract level carry over to the concrete system. As an example, we presented an abstract specification and a concrete implementation of secure message transmission with ordered channels and we formally verified both the security of this concrete implementation and its actual goals.

Chapter 6

Polynomial Fairness and Liveness

After introducing integrity properties in the precedent chapter, we now focus on introducing what fairness and liveness for arbitrary cryptographic protocols means. Unfortunately, we will see that the common definitions of fairness and liveness are not suited to cope with real cryptography, so we present new definitions which we denote by *polynomial fairness* and *polynomial liveness*. Similar to integrity properties in the previous chapter, we show that these liveness properties are maintained under simulatability. As an example fitting our definition, we present an abstract specification and a concrete implementation of secure message transmission with reliable channels. Here, reliability is considered as the desired liveness property, i.e., roughly speaking, every message sent over these channels will eventually be delivered. We conclude with a short summary as usual.

6.1 Introduction and Related Literature

If we consider properties of arbitrary protocols, we can distinguish between liveness and safety properties (according to the characterization of Alpern and Schneider [5]), besides confidentiality properties. Informally, a liveness property states that something good eventually happens. One usual problem in asynchronous scenarios is that liveness depends on the scheduler: if the scheduler never schedules certain messages, these messages will not proceed and nothing good will ever happen. The standard solution (see, e.g., [32]) is to concentrate on so-called fair schedulers. Roughly, those guarantee that every message is delivered at some point of time in every infinite run of the system.

For most cryptographic protocols, these definitions cannot be used because one must restrict the adversary and the runs as a whole to polynomial length; hence, the notion of infinity does not apply. Another problem in the cryptographic case is that one typically assumes that an adversary can modify messages arbitrarily in transit. For those cases, one can certainly not require that anything good happens (e.g., the parties agree on a key [46]). Nevertheless, even for protocols that do consider such arbitrary corruptions, one typically wants a guarantee of the following form: *If* certain messages get through unmodified, then certain good things happen. For the other cases, e.g., the honest participants make progress but do not get each others' messages, one may still

want at least local termination (in the sense of a timeout message, or a positive reaction to an abort by the user).

In order to cope with the above problems, we introduce the notion of *polynomial fairness* and *polynomial liveness*. Roughly, polynomial fairness states that the master scheduler will schedule each message after a polynomially bounded number of steps. Now, polynomial liveness captures that something good will happen after a polynomial number of steps of the honest user subject to the condition that the considered scheduler is polynomially fair.

We are only aware of one paper that handles polynomial liveness properties for security protocols: Cachin et al. presented a nice approach for a specific protocol, asynchronous Byzantine agreement in [11]. It is shown that an adversary cannot make the honest users (altogether) generate a super-polynomial number of messages for any particular subprotocol run and that the protocol ensures "deadlock-freeness", i.e., some progress will eventually be made. However, their definition was limited to this specific protocol, and applying the approach more generally presumes a fixed number of subprotocols and the use of session ID's (so that one can say "all messages associated to an event have been delivered"). We aim at a more general definition. Thus, the fact that certain messages get through is defined by letting the system "run empty", i.e., we consider one particular point in time, so that neither the honest user nor the adversary produce any outputs after that time. From then on, the only active machines are the scheduler and the internal machines of the system, so the scheduler can deliver all messages that have already passed through the adversary (i.e., have not been interrupted in transit), all messages that have been sent over reliable channels etc. We speak of polynomial liveness if the good event then happens in a polynomially bounded number of steps.

To the best of our knowledge, the approach of letting the system run empty has not been used before to prove arbitrary properties of security protocols. However, if we prescind from the security community and take a look at verification of microprocessors we will meet similar techniques. Roughly speaking, Burch and Dill showed in [10] that certain safety properties of a pipelined microprocessor hold by letting the pipelines run empty now and then, which they denoted as "flushing the pipeline". In the essence, their approach is quite similar to ours, but applied to a completely different problem.

In this work, we only consider one run-empty phase. In principle, the user and adversary could restart outputting messages after the good event has happened, so that liveness would be extended to multiple good events. However, this task is tedious and considered as future work.

Moreover, we will show that our definition of liveness behaves well under the concept of simulatability which has asserted its position as a fundamental concept of cryptography. Precisely, we show that liveness properties are preserved under simulatability under certain circumstances, i.e., liveness properties proved for abstract specifications automatically carry over to the concrete implementations in this case.

As an example fitting our definition, we present an abstract specification and a concrete implementation of reliable secure message transmission. The reliability is considered as the desired liveness property, i.e., roughly speaking, every message sent will eventually be delivered. We prove this liveness property for the abstract specification and transfer it to the implementation with the preservation theorem.

6.2 Expressing Polynomial Fairness and Liveness

In this section we introduce our definitions of polynomial fairness and polynomial liveness in asynchronous reactive systems. At first, we concentrate on fairness.

6.2.1 Polynomial Fairness

Usually, a scheduler is called *fair* if it schedules every process infinitely often unless this process is only finitely often enabled. As we already stated in the introduction, this definition is not suited for schedulers of most cryptographic protocols, since both the adversary and the honest user are polynomially bounded, and the runs as a whole are restricted to polynomial length. Hence, the terminology of infinity does not apply.

What we would like to express is that an enabled process will always be scheduled by the master scheduler after a polynomially bounded number $J(k)$ of the master scheduler's steps unless the master scheduler reaches its runtime bound. We start with an intuitive description of how this can be formalized.

Starting from the t-th view-step of X in one particular trace tr (denoted by $O_{t,tr}$), we search for the first future time $m > t$ such that the first message of $\tilde{\mathsf{p}}$ is scheduled (denoted by $(O_{m,tr})_{\mathsf{p}^{\triangleleft}!} = 1$). Thus, if the buffer is non-empty, a message will be scheduled from it. Moreover, we always demand that it is the first clock-out port with non-empty value (so it will not be ignored by the run algorithm, cf. Definition 2.6). We denote this number of view-steps (i.e., $m - t$) by $\mathsf{wait}(t, tr, \mathsf{p}^{\triangleleft}!)$. Moreover, in order to cope with the final state of X, we explicitly define this number to be infinite if the master scheduler never enters a final state, and if there exists no such output at $\mathsf{p}^{\triangleleft}!$. If the master scheduler enters a final state after its m-th view-step without ever outputting 1 at $\mathsf{p}^{\triangleleft}!$ we define this number to be $m - t$.

We denote the scheduler as J-fair for a function $J : \mathbb{N} \to \mathbb{N}$ if the maximum of these waiting times is bounded by J as a function of k.

Moreover, we demand that X does not connect to an unspecified port of the system, i.e., we have $\mathsf{ports}(\mathsf{X}) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$. This condition is essential for relating the definition of polynomial fairness to simulatability, since the master scheduler can be defined as part of the honest user in this case, and hence, can remain unchanged in simulatability.

Let us now turn to the formal definition.

Definition 6.1 *(Polynomial Fairness) Let an arbitrary system Sys be given. Then a master scheduler X is called* J-fair *for a structure $(\hat{M}, S) \in Sys$ if and only if the following holds:*

- *X does not connect to an unspecified port of the system, i.e., we have $\mathsf{ports}(\mathsf{X}) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$.*
- *Let $O_{t,tr}$ and $S_{t,tr}$ be X's output and state after the t-th switching of X, respectively, in every considered trace tr of the configuration. Moreover, let Fin_{X} denote the set of final states of X. Now we define*

$$\mathsf{wait}(t, tr, \mathsf{p}^{\triangleleft}!) := \infty$$

if for all $m \in \mathbb{N}$: $(O_{m+t,tr})_{\mathsf{p}^{\triangleleft}!} \neq 1$ and $S_{m+t,tr} \notin Fin_{\mathsf{X}}$, and otherwise

$$\mathsf{wait}(t, tr, \mathsf{p}^{\triangleleft}!) := \mathsf{min}_{m \in \mathbb{N}}\{(O_{m+t,tr})_{\mathsf{p}^{\triangleleft}!} = 1 \vee S_{m+t,tr} \in Fin_{\mathsf{X}}\}.$$

Then we require that

$$\mathsf{wait}(t, tr, \mathsf{p}^{\triangleleft}!) \leq J(k)$$

holds for all traces tr of the configuration, all $t \in \mathbb{N}$ and $\mathsf{p}^{\triangleleft}! \in \mathsf{ports}(\mathsf{X})$.

If a master scheduler is J-fair for a polynomial J, we call it polynomially fair. *In the following, we will augment a master scheduler X by the index $_{\mathsf{fair}}$ if it is fair, i.e., we write $\mathsf{X}_{\mathsf{fair}}$.* ◇

Remark 6.1. Our definition of fairness implies the common definition of fairness based on infinite sequences (i.e., processes which are infinitely often enabled have to be scheduled infinitely often). By "a process or machine M is enabled", we define that an ingoing buffer $\widetilde{\mathsf{p}}$ of the machine M has non-empty content, and the master scheduler could schedule this buffer at the moment (i.e., the master scheduler is switched, and it has the corresponding clockout port $\mathsf{p}^{\triangleleft}!$ for scheduling this buffer). Now according to our definition, this buffer will always be scheduled after at most $J(k)$ steps for a polynomial J and the security parameter k, i.e., after a finite number of steps. Moreover, if the machine is enabled infinitely often, then the master scheduler must also be switched infinitely often, so it will schedule the corresponding buffer, and therefore the machine infinite times. ◦

6.2.2 Polynomial Liveness

After introducing the notion of polynomially fair master schedulers, we can now turn our attention to expressing polynomial liveness. Intuitively speaking, polynomial liveness means that some good things will happen after a polynomial number of steps of the honest user.

However, we cannot expect this to hold for arbitrary configurations. Imagine an honest user and an adversary which are communicating over the system all the time, without ever giving control to the (fair) master scheduler. In this case we cannot guarantee that good things happen since the fairness condition of the master scheduler is irrelevant. Instead, we define that certain good things happen if the system is "run empty". More precisely, we consider situations where neither the user nor the adversary produce any outputs any further from one particular point in time. We then require that there exists a polynomial Q_{H} such that the good event will happen in at most $Q_{\mathsf{H}}(k)$ view-steps of H, counted from that special point in time.

Thus, in order to define polynomial liveness for the overall system, we restrict ourselves to those configurations which prevent outputs of both the user and the adversary after a particular number of view-steps of H. More precisely, we let the honest user count the number of times it has been switched. If it reaches the "critical" time $t_{\mathsf{stop}}(k)$, it outputs a command (stop) to the adversary and both machines do not produce outputs any further. We call these configurations *liveness configurations*.

From that special point in time, the master scheduler and the machines of the system are the only active machines in the configuration, so the master scheduler can try to empty the system, e.g., to deliver those messages that have already passed through the adversary and now wait to be scheduled to their recipient, and those that have been sent over reliable channels.

However, there is one more problem we have to take care of. Clearly, we cannot expect the event to happen if the master scheduler or the honest user enter a final state too early. Therefore, we assume that the master scheduler and the honest user run sufficiently long, i.e., we augment our definition of liveness configurations with lower

bounds Q_{X}, Q_{H} on the number of view-steps of the master scheduler and the honest user.

Definition 6.2 *(Liveness Configuration) Let an arbitrary system Sys and four functions $t_{\mathsf{stop}}, J, Q_{\mathsf{X}}, Q_{\mathsf{H}} \colon \mathbb{N} \to \mathbb{N}$ be given. Furthermore, let a configuration $conf = (\hat{M}, S, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A}) \in \mathsf{Conf}(Sys)$ be given where $\mathsf{X}_{\mathsf{fair}}$ is a J-fair scheduler for $(\hat{M}, S)$. We call this configuration a (t_{stop}, J, Q_{X}, Q_{H})-*liveness configuration *if the following holds:*

- *The honest user H has special ports $stop_{\mathsf{H}}!$, $stop_{\mathsf{H}}{}^{\triangleleft}!$ which are connected to the adversary, i.e., $\{stop_{\mathsf{H}}!, stop_{\mathsf{H}}{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{H})$ and $stop_{\mathsf{H}}? \in \mathsf{ports}(\mathsf{A})$.*
- *The user H has an internal counter $count$ over the naturals initialized with 0. The user first increases this counter every time it switches, and then checks whether the counter equals t_{stop} (k). In this case it outputs (stop) at $stop_{\mathsf{H}}!$, 1 at $stop_{\mathsf{H}}{}^{\triangleleft}!$. From now on, the user only reads its inputs, but no longer produces outputs. Similarly, if the adversary gets an input (stop) at $stop_{\mathsf{H}}?$, it only reads its inputs in the future without producing any outputs.*
- *The user H does not output anything at $stop_{\mathsf{H}}!$ except in the above case.*
- *The number of view-steps of the master scheduler and the honest user is lower-bounded by $Q_{\mathsf{X}}(k)$ and $Q_{\mathsf{H}}(k)$, respectively, for every trace of the configuration.*

Liveness configurations will be denoted by $conf^{\mathsf{live}} = (\hat{M}, S, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A})^{\mathsf{live}}$, the set of all liveness configurations of a system Sys by $\mathsf{Conf}^{\mathsf{live}}(Sys)$, and the set of all polynomial-time liveness configurations by $\mathsf{Conf}^{\mathsf{live}}_{\mathsf{poly}}(Sys)$. ◇

Thus, for a given trace tr of a (t_{stop}, J, Q_{X}, Q_{H})-liveness configuration, we will not have any further outputs of both the user and the adversary after the $t_{\mathsf{stop}}(k)$-th switching of the user. We use this point in time to split the trace into two parts, a prefix tr_i and a tail $extr_i$, so that we obtain $tr = tr_i \circ extr_i$. The tail $extr_i$ is called the *extended trace* or the *run-empty phase*.

Remark 6.2. If the user has exceeded the time $t_{\mathsf{stop}}(k)$ he will never be able to produce any outputs again. In real life, the user will usually resume outputs after the good event has happened, e.g., send reply-messages. Our definition could be modified such that both the honest user and the adversary are "switched on" again after the good event has happened. However, this task is tedious because it significantly complicates our upcoming definitions and the preservation theorem of Section 6.3, so we will consider it as future work only. ◦

We can now turn our attention to the actual definition of polynomial liveness.

Definition 6.3 *(Polynomial Liveness Properties) A polynomial liveness property of a structure $(\hat{M}, S) \in Sys$ consists of three components:*

1. *First, we have an integrity property Req (the good event which we would like to happen), cf. Definition 5.1, i.e., Req is a function that assigns a set of traces at the ports in S to each set S with $(\hat{M}, S) \in Sys$. Informally speaking, $Req(S)$ states which are the "good" traces for the given structure $(\hat{M}, S)$.*

2. *The second component* $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}$ *is a subset of the complement of the specified ports of the structure, i.e.,* $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq S^c$*. It indicates which ports have to be scheduled by the fair master scheduler such that the event* Req *will eventually happen.*

3. *The third component is a function* $t_s \colon \mathbb{N} \to \mathbb{N}$*. Intuitively, a system can only run empty and finally fulfill the desired property if it has not yet run for too long. In this case, it might exceed its runtime bounds during the extended trace of the configuration before the event* Req *occurs. Therefore, we have to bound the point in time at which the extended trace may begin. Obviously, the runtime of the system depends on the security parameter* k*, so this bound is represented as a function of* k *too. Intuitively, the function* t_s *denotes that the event* Req *will happen if we restrict our attention to the set of all (*t_{stop}*,* J*,* Q_{X}*,* Q_{H}*)-liveness configurations for* $t_{\mathsf{stop}}(k) < t_s(k)$*, for every polynomial* J *and suitable choices for* Q_{X} *and* Q_{H}*.*

Finally, a liveness property of a system Sys *is a mapping*

$$\begin{aligned} \varphi_{Sys} : Sys &\to LiveProp \\ (\hat{M}, S) &\mapsto (Req, \{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s)_{(\hat{M},S)} \end{aligned}$$

that assigns each structure $(\hat{M}, S)$ *a liveness property which is defined on* $(\hat{M}, S)$*.* $\diamond$

After introducing what polynomial liveness properties are, we have to define what it means that a system fulfills them. Essentially, our complete definition states that a structure fulfills the liveness property $(Req, \{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s)$ if the following holds:

If the ports $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}$ are scheduled by a J-fair master scheduler for an arbitrary polynomial J, and if we do not proceed too far in time (i.e., we only consider (t_{stop},J,$\cdot$,$\cdot$)-liveness configurations for $t_{\mathsf{stop}}(k) < t_s(k)$) then there are polynomials Q_{X}, Q_{H} such that the event Req will happen within a polynomial number of view-steps of the honest user.

This will be expressed by using prefixes of the whole run restricted to those ports that connect to the honest user in the considered configuration, i.e., we write $run_{conf,k,l(k)}\lceil_{S^{\mathsf{H}}}$ with $S^{\mathsf{H}} := \{\mathsf{p} \in S \mid \mathsf{p}^c \in \mathsf{ports}(\mathsf{H})\}$ and a polynomial l (cf. Definition 2.7). In slight abuse of notation, we write $Req(S^{\mathsf{H}})$ instead of $Req(S)\lceil_{S^{\mathsf{H}}}$, i.e., we restrict the trace to the ports in S^{H}. A system fulfills the overall liveness property if all of its structures fulfill their liveness properties. Moreover, we will see that there are different grades of fulfillment. We distinguish between *perfect*, *statistical* and *computational* fulfillment depending on whether the good event will always happen, or only with overwhelming probability, i.e., the probability of failure should be statistically small or negligible in polynomial-time configurations, respectively.

Definition 6.4 *(Fulfillment of Polynomial Liveness Properties) Let an arbitrary system* Sys *and a polynomial liveness property* φ_{Sys} *for* Sys *be given. Then a structure* $(\hat{M}, S) \in Sys$ *fulfills its polynomial liveness property* $LiveReq := (Req, \{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s) := \varphi_{Sys}(\hat{M}, S)$

- ***perfectly*** *(*$(\hat{M}, S) \models_{\mathsf{perf}} LiveReq$*) iff* $\forall$ *polynomials* J, t_{stop} *with* $t_{\mathsf{stop}} < t_s$ $\exists$ *polynomials* Q_{X}*,* Q_{H} *such that for all (*t_{stop}*,* J*,* Q_{X}*,* Q_{H}*)-liveness configurations*

$conf_{\mathsf{live}} = (\hat{M}, S, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A})^{|\mathsf{ive}} \in \mathsf{Conf}^{|\mathsf{ive}}(Sys)$ with $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{X}_{\mathsf{fair}})$ the following holds: all $Q_{\mathsf{H}}(k)$-prefixes of the restriction of the run to the ports in S^{H} lie in $Req(S^{\mathsf{H}})$. In formulas,

$$[(run_{conf_{\mathsf{live}}, k, Q_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}})] \subseteq Req(S^{\mathsf{H}})$$

for all k, where $[\cdot]$ denotes the carrier set of a probability distribution.

- ***statistically*** *($(\hat{M}, S) \models_{SMALL} LiveReq$) iff $\forall$ polynomials J, t_{stop} with $t_{\mathsf{stop}} < t_s$ $\exists$ polynomials Q_{X}, Q_{H} such that for all (t_{stop}, J, Q_{X}, Q_{H})-liveness configurations $conf_{\mathsf{live}} = (\hat{M}, S, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A})^{|\mathsf{ive}} \in \mathsf{Conf}^{|\mathsf{ive}}(Sys)$ with $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{X}_{\mathsf{fair}})$ the following holds: the probability that $Req(S^{\mathsf{H}})$ is not fulfilled after $Q_{\mathsf{H}}(k)$ steps is small, i.e.,*

  $$P(run_{conf_{\mathsf{live}}, k, Q_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}})) \in SMALL.$$

  The class $SMALL$ must be closed under addition and making functions smaller.

- ***computationally*** *($(\hat{M}, S) \models_{\mathsf{poly}} LiveReq$) iff $\forall$ polynomials J, t_{stop} with $t_{\mathsf{stop}}(k) < t_s(k)$ $\exists$ polynomials Q_{X}, Q_{H} such that for all polynomial-time (t_{stop}, J, Q_{X}, Q_{H})-liveness configurations with $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq \mathsf{ports}(\mathsf{X}_{\mathsf{fair}})$ the following holds: the probability, that $Req(S^{\mathsf{H}})$ is not fulfilled after $Q_{\mathsf{H}}(k)$ steps is negligible, i.e.,*

  $$P(run_{conf_{\mathsf{live}}, k, Q_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}})) \in NEGL.$$

We write $(\hat{M}, S) \models (Req, \{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s)$ if we want to treat all three cases together.

Finally, a system Sys fulfills a liveness property φ_{Sys} perfectly, statistically, or computationally iff each $(\hat{M}, S) \in Sys$ fulfills $\varphi_{Sys}(\hat{M}, S)$ perfectly, statistically, or computationally. In this case, we write $Sys \models_{\mathsf{perf}} \varphi_{Sys}$, etc. ◇

6.3 Preservation of Polynomial Liveness under Simulatability

In this section we show that our definition of polynomial liveness behaves well under simulatability under certain circumstances. Usually, defining a cryptographic system starts with an abstract specification stating what the system should do. After that, this specification can be refined stepwise with respect to simulatability, which finally yields a secure implementation. At this time, we may wonder whether or not the verification of these properties made for the ideal specification carries over to the concrete implementation. This is essential for modular proofs. We can answer this question in the affirmative under reasonable assumptions yielding the preservation theorem presented below.

In the following, we assume two systems Sys_1, Sys_2 to be given, such that $Sys_1 \geq^f Sys_2$ holds for a valid mapping f. Moreover, we assume that Sys_2 fulfills an arbitrary liveness property φ_{Sys_2}. However, we cannot expect the liveness property to automatically carry over to Sys_1 if both systems are completely unrestricted for

the following reason: assume that we have given a valid $(t_{\mathsf{stop}},\cdot,\cdot,\cdot)$-liveness configuration $conf_1^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys_1)$, so there has to be an indistinguishable configuration $conf_2 \in \mathsf{Conf}(Sys_2)$ because of $Sys_1 \geq^f Sys_2$. However, it is clear that $conf_2$ is not necessarily a $(t_{\mathsf{stop}},\cdot,\cdot,\cdot)$-liveness configurations again, since the adversary is not forced to stop outputting messages at that particular point in time; in fact, he is not forced to do so at all. The remaining three parameters are omitted because they must remain unchanged under simulatability since they are part of the honest user.

Hitting the spot, simulatability is not forced to map liveness configurations of the first system to liveness configurations of the second. Thus, given an arbitrary $(t_{\mathsf{stop}},\cdot,\cdot,\cdot)$-liveness configuration $conf_1^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys_1)$, we cannot exploit our assumption $Sys_2 \models \varphi_{Sys_2}$ by means of simulatability in order to decide whether or not the considered liveness property holds for the first system. We therefore have to restrict our attention to those systems in which simulatability respects $(t_{\mathsf{stop}},\cdot,\cdot,\cdot)$-liveness configurations, i.e., simulatability yields indistinguishable $(t_{\mathsf{stop}},\cdot,\cdot,\cdot)$-liveness configurations by construction. We speak of *liveness simulatability* in this case.

At first glance, this seems to be a quite severe restriction to the considered set of possible systems. However, indistinguishable configurations are typically derived using the simulatability variant of *blackbox* simulatability. This means that the adversary A' of the indistinguishable configuration is derived by the original adversary A and a simulator Sim which is inserted between the original adversary and the system. This does not change the communication between A and the honest user, so A' handles incoming (stop)-signals just as the original adversary A. Moreover, the machine Sim usually only transmits values from the adversary to the system and vice versa; especially it does not produce any outputs alone by itself. Thus, the complete adversary A', i.e., A and Sim, will not produce outputs any further if the original adversary does not, yielding the desired liveness configuration. Our example of Section 6.4 belongs to that kind of system. All other examples which have been proved so far, e.g., secure channels, fair exchange protocols, and secure group key exchange, belong to that kind of system as well. Formally, liveness simulatability is introduced as follows.

Definition 6.5 *(Liveness Simulatability) Let two arbitrary systems Sys_1 and Sys_2 be given such that $Sys_1 \geq^f Sys_2$ holds for a valid mapping f. We then call Sys_1 "at least as secure as" Sys_2 "with respect to liveness" (written $Sys_1 \geq^{f,\mathsf{live}} Sys_2$) if the following holds: for a given $(t_{\mathsf{stop}}, J, Q_{\mathsf{X}}, Q_{\mathsf{H}})$-liveness configuration $conf_1^{\mathsf{live}} = (\hat{M}_1, S_1, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A}_1)^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys_1)$ there exists a $(t_{\mathsf{stop}}, J, Q_{\mathsf{X}}, Q_{\mathsf{H}})$-liveness configuration $conf_2^{\mathsf{live}} = (\hat{M}_2, S_2, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A}_2)^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys_2)$ yielding indistinguishable views for the honest user. As usual, we distinguish between perfect, statistical, and computational indistinguishability.* ◇

We will now show that liveness properties automatically carry over in case of liveness simulatability, if the important ports of the master scheduler for achieving liveness are identical in both the considered ideal and real system.

Theorem 6.1 *(Preservation of Polynomial Liveness) Let an arbitrary system Sys_2 and a polynomial liveness property φ_{Sys_2} be given such that $Sys_2 \models \varphi_{Sys_2}$ holds. Furthermore, let a system Sys_1 be given with $Sys_1 \geq^{f,\mathsf{live}} Sys_2$ for a mapping f with $S_1 = S_2$ whenever $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$. Then $Sys_1 \models \varphi_{Sys_1}$ for all φ_{Sys_1} with $\varphi_{Sys_1}(\hat{M}_1, S_1) := \varphi_{Sys_2}(\hat{M}_2, S_2)$ for an arbitrary structure $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$.*

This holds in the perfect case and in the statistical case. It holds in the computational case if additionally, membership in $Req(S)$ is decidable in polynomial time for

all S. □

Proof. At first, we show that φ_{Sys_1} is a well-defined liveness property for Sys_1. Let an arbitrary structure $(\hat{M}_1, S_1) \in Sys_1$ be given. Simulatability implies that for every structure $(\hat{M}_1, S_1) \in Sys_1$ there exists $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$. φ_{Sys_2} is a well-defined liveness property for Sys_2, so $\varphi_{Sys_2}(\hat{M}_2, S_2) =: (Req, \{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s)$ is a liveness property for $(\hat{M}_2, S_2)$. By precondition, we have $S_1 = S_2$, so the integrity requirement Req is well-defined on $(\hat{M}_1, S_1)$. Moreover, we have $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq S_1^c$ if and only if $\{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\} \subseteq S_2^c$. Therefore, the liveness property $\varphi_{Sys_1}(\hat{M}_1, S_1)$ is defined on the structure $(\hat{M}_1, S_1)$, so φ_{Sys_1} is a well-defined liveness property of Sys_1. We now have to show that Sys_1 fulfills φ_{Sys_1}. The actual proof will be done by contradiction, i.e., we will show that if Sys_1 would not fulfill the liveness property, the two systems could be distinguished.

Assume that Sys_1 does not fulfill its liveness property. Thus, there exist polynomials J, t_{stop} with $t_{\mathsf{stop}} < t_s$ such that for every polynomials Q_{X}, Q_{H} there exists a (t_{stop}, J, Q_{X}, Q_{H})-liveness configuration $conf_1^{\mathsf{live}}$ so that the good event does not occur within $Q_{\mathsf{H}}(k)$ view-steps of the honest user. Because of $Sys_1 \geq^{f,\mathsf{live}} Sys_2$ there is a (t_{stop}, J, Q_{X}, Q_{H})-liveness configuration $conf_2^{\mathsf{live}} = (\hat{M}_2, S_2, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A}_2)^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys_2)$ for $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ and for every polynomial Q_{X} and Q_{H} such that

$$view_{conf_1^{\mathsf{live}}}(\{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}) \approx view_{conf_2^{\mathsf{live}}}(\{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\})$$

holds. For the sake of readability we abbreviate $\{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}$ by H' and set $S := S_1 := S_2$. Since H is a submachine of H', we can apply Lemma 2.4 which yields $view_{conf_1^{\mathsf{live}}}(\mathsf{H}) \approx view_{conf_2^{\mathsf{live}}}(\mathsf{H})$. Moreover, the view of H in both configurations contains the trace at S^{H}, i.e., the trace is a function of the view, so we finally obtain

$$run_{conf_1^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}} \approx run_{conf_2^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}}.$$

As usual we have to distinguish between the perfect, statistical and computational case. In the computational case, both configurations have to be polynomial-time.

In the perfect case we have $view_{conf_1^{\mathsf{live}}}(\mathsf{H}') = view_{conf_2^{\mathsf{live}}}(\mathsf{H}')$ because of $Sys_1 \geq^{f,\mathsf{live},\mathsf{perf}} Sys_2$, i.e., the distributions of the views are identical which yields $run_{conf_1^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}} = run_{conf_2^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}}$. Because of $(\hat{M}_2, S_2) \models_{\mathsf{perf}} (Req, \{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s)$, there exist two polynomials Q'_{X}, Q'_{H} such that

$$[(run_{conf_2^{\mathsf{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}})] \subseteq Req(S^{\mathsf{H}})$$

holds for every (t_{stop}, J, Q'_{X}, Q'_{H})-liveness configuration $conf_2^{\mathsf{live}}$. Thus, for every given (t_{stop}, J, Q'_{X}, Q'_{H})-liveness configuration $conf_1^{\mathsf{live}}$, we have an indistinguishable (t_{stop}, J, Q'_{X}, Q'_{H})-liveness configuration $conf_2^{\mathsf{live}}$ of $(\hat{M}_2, S_2)$ with the above property. Assume now that $(\hat{M}_1, S_1)$ does not fulfill $(Req, \{\mathsf{p}_1{}^{\triangleleft}!, \ldots, \mathsf{p}_n{}^{\triangleleft}!\}, t_s)$. This immediately contradicts the assumption that $[(run_{conf_1^{\mathsf{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}})] \not\subseteq Req(S^{\mathsf{H}})$ while $[(run_{conf_2^{\mathsf{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}})] \subseteq Req(S^{\mathsf{H}})$, since $run_{conf_1^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}} = run_{conf_2^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}}$ holds.

In the statistical case, we have $view_{conf_1^{\mathsf{live}}}(\mathsf{H}') \approx_{SMALL} view_{conf_2^{\mathsf{live}}}(\mathsf{H}')$, which again yields $run_{conf_1^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}} \approx_{SMALL} run_{conf_2^{\mathsf{live}}}\lceil_{S^{\mathsf{H}}}$. Thus, the statistical distance $\Delta(run_{conf_1^{\mathsf{live}},k,l(k)}\lceil_{S^{\mathsf{H}}}, run_{conf_2^{\mathsf{live}},k,l(k)}\lceil_{S^{\mathsf{H}}})$ is a function $g(k) \in SMALL$ for all polynomials l. We apply Lemma 2.4 to the characteristic function $1_{v\lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}})}$ on such

views v. This gives

$$\begin{aligned} & |P(run_{conf_1^{\mathrm{live}},k,l(k)}\lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}})) \\ - \quad & P(run_{conf_2^{\mathrm{live}},k,l(k)}\lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}}))| \\ \leq \quad & g(k). \end{aligned}$$

for every polynomial l. If we use the above inequality with $l := Q'_{\mathsf{H}}$ we obtain

$$\begin{aligned} & |P(run_{conf_1^{\mathrm{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}})) \\ - \quad & P(run_{conf_2^{\mathrm{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}}))| \\ \leq \quad & g(k). \end{aligned}$$

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 as follows: Given a view of machine H, it extracts the $Q'_{\mathsf{H}}(k)$ prefix of the user's view restricted to S^{H} and verifies if the result lies in $Req(S^{\mathsf{H}})$. 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 $Req(S)$ (and therefore also in $Req(S^{\mathsf{H}})$) was required to be polynomial-time decidable. Its advantage in distinguishing is

$$\begin{aligned} & |P(\mathsf{Dis}(1^k, view_{conf_1^{\mathrm{live}},k}) = 1) \\ - \quad & P(\mathsf{Dis}(1^k, view_{conf_2^{\mathrm{live}},k}) = 1)| \\ = \quad & |P(run_{conf_1^{\mathrm{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}})) \\ - \quad & P(run_{conf_2^{\mathrm{live}},k,Q'_{\mathsf{H}}(k)}\lceil_{S^{\mathsf{H}}} \notin Req(S^{\mathsf{H}}))|. \end{aligned}$$

If this difference were negligible, then the first term would have to be negligible because the second term is and $NEGL$ is closed under addition. Again this is the desired contradiction. ∎

6.4 An Example: Secure Message Transmission with Reliable Channels

In the following we present a specification for secure message transmission with reliable channels. Here, reliability is considered as a liveness property which the system will be proved to fulfill. Moreover, we present a secure implementation.

Both the ideal and real system are based on the systems for secure message transmission which we reviewed in Section 2.3. However, we will have to modify them to fit our requirements.

6.4.1 The Ideal System

Recall that the ideal system of secure message transmission is of the typical form

$$Sys_{\mathsf{id}} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) | \mathcal{H} \in \mathcal{ACC}\},$$

and $\mathcal{ACC}$ is the powerset of $\{1, \ldots, n\}$, where n denotes the number of possible participants. The system is illustrated at the left side of Figure 6.1.

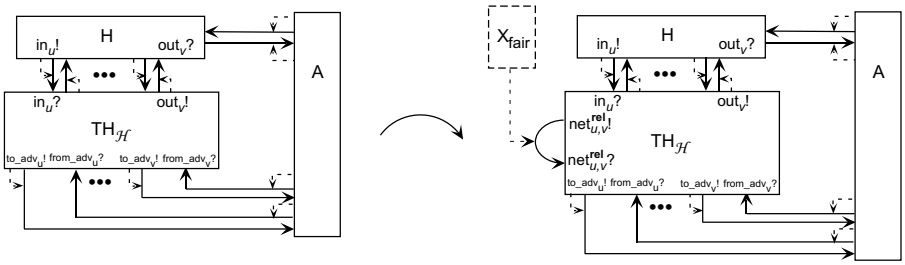

Figure 6.1: The left part shows the unmodified ideal system Sys_{id}, the right part shows the modified system $Sys^{\mathsf{rel}}_{\mathsf{id}}$. Exemplarily, we only plotted one self-loop port $\mathsf{net}^{\mathsf{rel}}_{u,v}$, and we sketched that this port is scheduled by the master scheduler $\mathsf{X}_{\mathsf{fair}}$.

Now, we want to modify the machine $\mathsf{TH}_{\mathcal{H}}$ such that it fulfills some kind of reliability, i.e., messages which have been sent over these reliable channels will eventually be delivered, and similarly for initialization. This can be achieved as follows.

Necessary Modification of the System. Roughly speaking, we model reliable channels by providing the trusted host with additional self-loop channels $\mathsf{aut}^{\mathsf{rel}}_{u,v}$ and $\mathsf{net}^{\mathsf{rel}}_{u,v}$, modeling the "reliable net" in the real world. The channels $\mathsf{aut}^{\mathsf{rel}}_{u,v}$ will be used for key exchange, i.e., they will be authenticated and reliable in the concrete implementation. The channels $\mathsf{net}^{\mathsf{rel}}_{u,v}$ will be used for message transmission, i.e., these channels will be regarded as reliable, but non-authenticated, so they can additionally be used by the adversary. This is illustrated at the right side of Figure 6.1.

More precisely, we assume a set $\mathcal{I}_{\mathcal{H}} \subseteq \mathcal{H}^2$ of pairs of users where $(u,v) \in \mathcal{I}_{\mathcal{H}}$ states that there is a reliable channel for sending messages from user u to user v. If we take a look at cryptographic applications in the real world the reader would perhaps consider symmetry as a necessary requirement for the relation $\mathcal{I}_{\mathcal{H}}$ since reliable channels are normally usable in both directions. However, we do not restrict ourselves to this special case; just imagine satelliting encrypted messages or sending them over radio channels. Intuitively speaking, those possibilities of communications would be regarded as reliable, but communication is limited to one direction since the recipient cannot simply send its response back via satellite.

Coming back to our specification, we model reliable channels for every pair $(u,v) \in \mathcal{I}_{\mathcal{H}}$ by providing the trusted host with additional self-loop channels $\{\mathsf{aut}^{\mathsf{rel}}_{u,v}!, \mathsf{aut}^{\mathsf{rel}}_{u,v}?\} \subseteq \mathsf{ports}(\mathsf{TH}_{\mathcal{H}})$ for key exchange and self-loop channels $\{\mathsf{net}^{\mathsf{rel}}_{u,v}!, \mathsf{net}^{\mathsf{rel}}_{u,v}?\} \subseteq \mathsf{ports}(\mathsf{TH}_{\mathcal{H}})$ for usual message transmission. The corresponding clock ports $\mathsf{aut}^{\mathsf{rel}}_{u,v}{}^{\triangleleft}!$ and $\mathsf{net}^{\mathsf{rel}}_{u,v}{}^{\triangleleft}!$ remain free at first; they will later be connected to the fair master scheduler to achieve the desired liveness property.

If now user u sends a command $(\mathsf{snd\_init})$ to $\mathsf{TH}_{\mathcal{H}}$, i.e., it wants to generate its keys, then $\mathsf{TH}_{\mathcal{H}}$ additionally outputs $(\mathsf{snd\_init})$ at $\mathsf{aut}^{\mathsf{rel}}_{u,v}!$ for all v with $(u,v) \in \mathcal{I}_{\mathcal{H}}$.

If user u sends a message to user v, $\mathsf{TH}_{\mathcal{H}}$ additionally outputs this (now blinded) message at $\mathsf{net}^{\mathsf{rel}}_{u,v}!$ if $(u,v) \in \mathcal{I}_{\mathcal{H}}$. In both cases, the usual output to the adversary remains unchanged, i.e., it still obtains the initialization request and the (blinded) message. By now, the blinded message or initialization command resides in the self-loop channel buffer and waits to be scheduled. If it is eventually scheduled by the master scheduler, the trusted host outputs the message to its recipient or initializes a connection, respectively. Intuitively, we can expect some kind of liveness property, since all

of these messages contained in the self-loops will eventually be delivered, at least if the master scheduler is fair and runs sufficiently long.

After presenting the main ideas, we can now turn our attention to the actual modifications of the system. Let $n \in \mathbb{N}$ and $\mathcal{M} := \{1, \ldots, n\}$ denote the number of participants and the set of indices, respectively, where $\mathcal{ACC}$ is the powerset of $\mathcal{M}$. Throughout the following, let an arbitrary $\mathcal{H} \in \mathcal{ACC}$ together with the set $\mathcal{I}_\mathcal{H}$ be given. As we already stated above, the standard trusted host $\mathsf{TH}_\mathcal{H}$ mainly has to be modified in the "Send" and "Send Initialization transitions. Moreover, we have to define what $\mathsf{TH}_\mathcal{H}$ does if it receives an input at one of the special new ports $\mathsf{aut}^{\mathsf{rel}}_{u,v}?$ or $\mathsf{net}^{\mathsf{rel}}_{u,v}?$ yielding two additional transitions of the machine. However, these transitions are quite close to the "Receive Initialization" and "Receive From Honest Party" transitions, so we decided to combine them. Thus, we obtain the following modifications:

- If $\mathsf{TH}_\mathcal{H}$ receives an input $(\mathsf{snd\_init})$ at port $\mathsf{in}_u?$, it additionally outputs $(\mathsf{snd\_init})$ at $\mathsf{aut}^{\mathsf{rel}}_{u,v}!$ for all v with $(u, v) \in \mathcal{I}_\mathcal{H}$. However, it still schedules the output intended for the adversary.
- If $\mathsf{TH}_\mathcal{H}$ receives an input (send, m, v) at port $\mathsf{in}_u?$, it checks whether $(u, v) \in \mathcal{I}_\mathcal{H}$. In this case it additionally outputs $(\mathsf{send\_blindly}, i, l, v)$ at $\mathsf{net}^{\mathsf{rel}}_{u,v}!$, but it still schedules the output intended for the adversary.
- If $\mathsf{TH}_\mathcal{H}$ receives an input $(\mathsf{snd\_init})$ at $\mathsf{aut}^{\mathsf{rel}}_{u,v}?$ it does the same checks as in the "Receive Initialization" transition, i.e., it checks that the machine of user v has not been stopped so far, that the connection between u and v has already been initialized and so on. If all these checks succeed, it outputs $(\mathsf{rec\_init}, u)$ at $\mathsf{out}_v!$.
- If $\mathsf{TH}_\mathcal{H}$ receives an input $(\mathsf{send\_blindly}, i, l, v)$ at $\mathsf{net}^{\mathsf{rel}}_{u,v}?$, it acts similarly as in the above case, i.e., it performs the test of the "Receive Message" transition, and if all tests succeed it outputs $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$.

After this rather informal definition which we hope to increase basic understanding, we now rigorously define our system. After that, we show how to modify the concrete implementation for secure message transmission in order to preserve the "at least as secure as" relation.

Scheme 6.1 (Secure Message Transmission with Reliable Channels) Let $n \in \mathbb{N}$ and a polynomial $L \in \mathbb{N}[x]$ be given. $L(k)$ bounds the length of the messages for the security parameter k. Let $\mathcal{M} := \{1, \ldots, n\}$ denote the set of possible participants, and let the access structure $\mathcal{ACC}$ be the powerset of $\mathcal{M}$. Moreover, let a family of sets $(\mathcal{I}_\mathcal{H})_{\mathcal{H} \in \mathcal{ACC}}$ be given such that $\mathcal{I}_\mathcal{H} \subseteq \mathcal{H} \times \mathcal{H}$. Our specification for secure message transmission with reliable channels is now a standard ideal system

$$Sys^{\mathsf{rel}}_{\mathsf{id}} = \{(\{\mathsf{TH}_\mathcal{H}\}, S_\mathcal{H}) \mid \mathcal{H} \in \mathcal{ACC}\}.$$

As in the standard, localized definition, we have $S^c_\mathcal{H} \supseteq \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$, but additionally, there are specified ports $\mathsf{net}^{\mathsf{rel}}_{u,v}{}^{\triangleleft}!$ and $\mathsf{aut}^{\mathsf{rel}}_{u,v}{}^{\triangleleft}!$ for every pair $(u, v) \in \mathcal{I}_\mathcal{H}$. Thus, we obtain

$$S^c_\mathcal{H} = \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{\mathsf{net}^{\mathsf{rel}}_{u,v}{}^{\triangleleft}!, \mathsf{aut}^{\mathsf{rel}}_{u,v}{}^{\triangleleft}! \mid (u, v) \in \mathcal{I}_\mathcal{H}\}.$$

The machine $\mathsf{TH}_\mathcal{H}$ is defined as follows. When $\mathcal{H}$ is clear from the context, let $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$ denote the indices of corrupted machines. The ports of

$\mathsf{TH}_{\mathcal{H}}$ are $\{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{\mathsf{net}^{\mathsf{rel}}_{u,v}!, \mathsf{net}^{\mathsf{rel}}_{u,v}? \mid (u,v) \in \mathcal{I}_{\mathcal{H}}\} \cup \{\mathsf{aut}^{\mathsf{rel}}_{u,v}!, \mathsf{aut}^{\mathsf{rel}}_{u,v}? \mid (u,v) \in \mathcal{I}_{\mathcal{H}}\} \cup \{\mathsf{from\_adv}_u?, \mathsf{to\_adv}_u!, \mathsf{to\_adv}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$ (cf. Figure 6.1).

$\mathsf{TH}_{\mathcal{H}}$ maintains arrays $(init^*_{u,v})_{u,v\in\mathcal{M}}$ and $(stopped^*_u)_{u\in\mathcal{H}}$ over $\{0,1\}$, both initialized with 0 everywhere, and an array $(deliver^*_{u,v})_{u,v\in\mathcal{H}}$ of lists, all initially empty. The state-transition function of $\mathsf{TH}_{\mathcal{H}}$ is defined by the following rules, written in the usual pseudo-code language.

Initialization.

- **Send initialization.** On input $(\mathsf{snd\_init})$ at $\mathsf{in}_u?$: If $stopped^*_u = 0$ and $init^*_{u,u} = 0$, set $init^*_{u,u} := 1$. After that, output $(\mathsf{snd\_init})$ at $\mathsf{aut}^{\mathsf{rel}}_{u,v}!$ for every $v \in \mathcal{H}$ with $(u,v) \in \mathcal{I}_{\mathcal{H}}$, $(\mathsf{snd\_init})$ at $\mathsf{to\_adv}_u!$, and 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$.
- **Receive initialization.** On input $(\mathsf{rec\_init}, u)$ at $\mathsf{from\_adv}_v?$ with $u \in \mathcal{M}, v \in \mathcal{H}$ or $(\mathsf{snd\_init})$ at $\mathsf{aut}^{\mathsf{rel}}_{u,v}?$: If $stopped^*_v = 0$ and $init^*_{u,v} = 0$ and $[u \in \mathcal{H} \Rightarrow init^*_{u,u} = 1]$, set $init^*_{u,v} := 1$ and output $(\mathsf{rec\_init}, u)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

Sending and receiving messages.

- **Send.** On input (send, m, v) at $\mathsf{in}_u?$ with $m \in \Sigma^+$, $l := \mathsf{len}(m) \leq L(k)$, and $v \in \mathcal{M} \setminus \{u\}$: If $stopped^*_u = 0$, $init^*_{u,u} = 1$, and $init^*_{v,u} = 1$: If $v \in \mathcal{A}$ then { output (send, m, v) at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$ } else $\{i := \mathsf{size}(deliver^*_{u,v}) + 1;\ deliver^*_{u,v}[i] := m.$ If $(u,v) \in \mathcal{I}_{\mathcal{H}}$ output $(\mathsf{send\_blindly}, i, l, v)$ at $\mathsf{net}^{\mathsf{rel}}_{u,v}!$, $(\mathsf{send\_blindly}, i, l, v)$ at $\mathsf{to\_adv}_u!$ and 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$. Otherwise, output $(\mathsf{send\_blindly}, i, l, v)$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$. }.
- **Receive from honest party u.** On input $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}_v?$ with $u, v \in \mathcal{H}$, $i \in \mathbb{N}$ or $(\mathsf{send\_blindly}, i, l, v)$ at $\mathsf{net}^{\mathsf{rel}}_{u,v}?$: If $stopped^*_v = 0$, $init^*_{v,v} = 1$, $init^*_{u,v} = 1$, and $m := deliver^*_{u,v}[i] \neq \downarrow$, then output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.
- **Receive from dishonest party u.** On input $(\mathsf{receive}, u, m)$ at $\mathsf{from\_adv}_v?$ with $u \in \mathcal{A}$, $m \in \Sigma^+$, $\mathsf{len}(m) \leq L(k)$, and $v \in \mathcal{H}$: If $stopped^*_v = 0$, $init^*_{v,v} = 1$ and $init^*_{u,v} = 1$, then output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.
- **Stop.** On input (stop) at $\mathsf{from\_adv}_u?$ with $u \in \mathcal{H}$, set $stopped^*_u = 1$ and output (stop) at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

$\diamond$

After presenting the abstract specification, we now show how to modify the concrete implementation for secure message transmission with ordered channels in order to preserve the "at least as secure as" relation.

6.4.2 The Real System

Recall that the concrete implementation of standard secure message transmission is a standard cryptographic system of the form

$$Sys_{\mathsf{real}} = \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \mathcal{H} \in \mathcal{ACC}\},$$

where $\mathcal{ACC}$ is the powerset of $\mathcal{M}$, i.e., any subset of participants may be dishonest. It uses asymmetric encryption and digital signatures as cryptographic primitives. A user u can let his machine create signature and encryption keys that are sent to other users over authenticated channels $\mathsf{aut}_{u,v}$. Furthermore, messages sent from user u to user v will be signed and encrypted by M_u and sent to M_v over an insecure channel $\mathsf{net}_{u,v}$, representing the net in the real world (cf. Figure 2.6 on page 24). The adversary is able to schedule the communication between the users, and he can furthermore send arbitrary messages m to arbitrary users u for a dishonest sender v.

We now have to implement the modification of the ideal system in this concrete implementation. In order to realize the authenticated reliable channels for key exchange between two users u and v, we simply define the clock port for clocking the channel $\mathsf{aut}_{u,v}$ as specified. Thus, the master scheduler can connect to it and we obtain a reliable and authenticated channel. Obviously, this modification yields the same functional behaviour as the modification in the ideal system.

The modification of the $\mathsf{net}_{u,v}$-channels is slightly more complicated. Obviously, we could simply define a new channel $\mathsf{net}^{\mathsf{rel}}_{u,v}$ between the machines M_u and M_v and let the master scheduler schedule it. However, this channel could only be used by the honest user, not by the adversary, so we would essentially assume a *secure* channel for message transmission. Thus, the whole cryptography could simply be omitted. Being more precise, the adversary could still send arbitrary messages to v, but they would arrive at port $\mathsf{net}^{\mathsf{a}}_{u,v}?$ of M_v, not at $\mathsf{net}^{\mathsf{rel}}_{u,v}?$. Thus, the machine M_v could distinguish between messages which have been sent by honest users (and which have not been modified), and between messages which have either been sent or modified by the adversary. Thus, we could as well omit the whole underlying cryptography, and let the machine decide whether the message should be accepted or not, so this kind of channels is not realistic in this case.

In order to circumvent this problem, we introduce so-called *reliable, non-authenticated* channels, i.e., channels which are reliable on the one hand, but additionally usable for the adversary on the other hand. They are defined as follows.

Definition 6.6 *(Reliable, Non-authenticated Channels) Let a structure $(\hat{M}, S)$ of a system Sys with $\hat{M} := \{\mathsf{M}_u \mid u \in \mathcal{H}\}$ be given. Consider a port $\mathsf{p} \in \mathsf{ports}(\mathsf{M}_u)$ with $\mathsf{p}^C \in \mathsf{ports}(\mathsf{M}_v)$ with $v \in \mathcal{H}$, i.e., $c = \{\mathsf{p}, \mathsf{p}^C\}$ is a high-level connection between two machines. If a channel model χ classifies this channel as* reliable non-authenticated *we modify the channel according to Figure 6.2, obtaining new machines $\mathsf{M}_{u,\mathcal{H}}$ and $\mathsf{M}_{v,\mathcal{H}}$.*

Without loss of generality, let p be an output port $\mathsf{p}!$. Then $\mathsf{M}_{u,\mathcal{H}}$ gets an additional new port $\mathsf{p}^{\mathsf{d}}!$ and its clock-out port $\mathsf{p}^{\mathsf{d}\triangleleft}!$, where it duplicates and immediately schedules outputs made at $\mathsf{p}!$. It is free so the adversary can connect to it. Additionally, the complement of the clockout port $\mathsf{p}^{\triangleleft}!$ is specified, i.e., $\mathsf{p}^{\triangleleft}! \in S^c$, so it can be scheduled by the master scheduler, which yields a reliable channel. The input port $\mathsf{p}?$ of $\mathsf{M}_{v,\mathcal{H}}$ is renamed into $\mathsf{p}^{\mathsf{r}}_{\mathsf{out}}?$. Furthermore, we define a machine $\check{\mathsf{p}}$ as follows:

The ports of $\check{\mathsf{p}}$ are given by $\{\mathsf{p}?, \mathsf{p}^{\mathsf{r}}_{\mathsf{in}}?, \mathsf{p}^{\mathsf{r}}_{\mathsf{out}}!, \mathsf{p}^{\mathsf{r}}_{\mathsf{out}}{}^{\triangleleft}!\}$, see Figure 6.2. On a non-empty input at either $\mathsf{p}?$ or $\mathsf{p}^{\mathsf{r}}_{\mathsf{in}}?$, $\check{\mathsf{p}}$ forwards this input at $\mathsf{p}^{\mathsf{r}}_{\mathsf{out}}!$ and schedules this input by outputting 1 at $\mathsf{p}^{\mathsf{r}}_{\mathsf{out}}{}^{\triangleleft}!$.

We assume without loss of generality that there is a systematic naming scheme for such new ports (e.g., appending $^{\mathsf{d}}$, $^{\mathsf{r}}_{\mathsf{out}}$, $^{\mathsf{r}}_{\mathsf{in}}$) that does not clash with prior names. ◇

Note, that message sent from either the honest user u or the adversary now arrive at the same port $\mathsf{p}^{\mathsf{r}}_{\mathsf{out}}?$ of machine $\mathsf{M}_{v,\mathcal{H}}$, if we use this kind of channels. Obviously, these

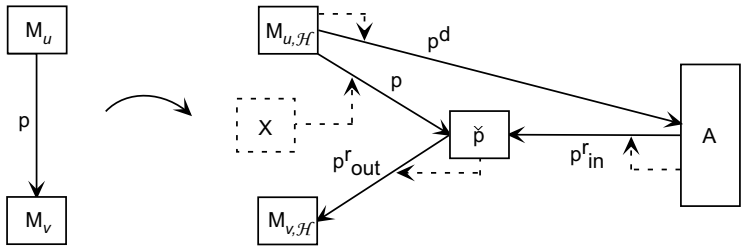

Figure 6.2: Modeling Reliable, Non-Authenticated Channels

reliable non-authenticated channels precisely model the modification of the trusted host $\mathsf{TH}_{\mathcal{H}}$ in the ideal system. We will denote the modified real system by $Sys^{\mathsf{rel}}_{\mathsf{real}}$.

Moreover, if we take a look at the security proof of [49] we can see that the concrete implementation is derived using blackbox simulatability, so our preservation theorem can be applied. Looking at the proof, it is quite obvious that this still holds for our modified systems.

6.4.3 Proof of Liveness

After introducing the specification, we now want to show that it in fact fulfills the desired liveness property, i.e., that messages sent by the honest user over reliable channels will eventually be received unless some internal checks of $\mathsf{TH}_{\mathcal{H}}$ fail (e.g., the user has not initialized itself, its machine has been stopped etc.).

At first, we have to rigorously define the liveness property $\varphi_{Sys^{\mathsf{rel}}_{\mathsf{id}}}$ which we aim to prove. Let an arbitrary structure $(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \in Sys^{\mathsf{rel}}_{\mathsf{id}}$ be given. Then the three components of $\varphi_{Sys^{\mathsf{rel}}_{\mathsf{id}}}(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}})$ are defined as follows.

- Its first component, the integrity requirement Req, is defined as follows. Consider two users $u, v, u \neq v$ such that $(u, v) \in \mathcal{I}_{\mathcal{H}}$. Informally, a trace is contained in the requirement if the following holds: if both users u and v have established a connection (i.e., they have initialized themselves, and both users have received the corresponding keys), and the user u has not been stopped so far, then a valid message sent from u to v will either be eventually delivered, or the recipient v has been or will be stopped.

  Formally, this is captured as follows. As usual, the lth step of the trace tr is denoted by tr_l. For the considered set $S_{\mathcal{H}}$ of specified ports, we define that a trace tr of an arbitrary configuration is contained in $Req(S_{\mathcal{H}})$ if the following holds.

  If there exists $l_1, l_2 \in \mathbb{N}$, such that

  $$\{\mathsf{in}_u!^c : (\mathsf{snd\_init}), \mathsf{in}_u{}^{\triangleleft}!^c : 1\} \subseteq tr_{l_1} \quad \text{(Key generation of } u \text{ in } tr_{l_1})$$
  $$\{\mathsf{in}_v!^c : (\mathsf{snd\_init}), \mathsf{in}_v{}^{\triangleleft}!^c : 1\} \subseteq tr_{l_2} \quad \text{(Key generation of } v \text{ in } tr_{l_2})$$

  and $l_3 > l_1, l_4 > l_2$ such that

  $$(\mathsf{out}_v?^c : (\mathsf{rec\_init}, u)) \in tr_{l_3} \quad \text{(Connection established from } u \text{ to } v \text{ in } tr_{l_3})$$
  $$(\mathsf{out}_u?^c : (\mathsf{rec\_init}, v)) \in tr_{l_4} \quad \text{(Connection established from } v \text{ to } u \text{ in } tr_{l_4})$$

then the following must hold. At first, set $l := \max\{l_3, l_4\}$. Then for every $t \in \mathbb{N}$,

$\{\mathsf{in}_u!^c : (\mathsf{send}, m, v), \mathsf{in}_u{}^{\triangleleft}!^c : 1\} \subseteq tr_t$	(User u sends a message m to v and schedules it)
$\wedge\ t > l$	(and a connection is established)
$\wedge\ m \in \Sigma^+ \wedge len(m) < L(k)$	(and the message is valid)
$\wedge\ \forall t_1 < t\colon (\mathsf{out}_u?^c : (\mathsf{stop})) \notin tr_{t_1}$	(and the machine of u has not been stopped so far)
$\Rightarrow\ (\exists t_2 \in \mathbb{N}\colon (\mathsf{out}_v?^c : (\mathsf{stop})) \in tr_{t_2}$	(then v's machine is either stopped in the run)
$\vee\ \exists t_3 > t\colon$	(or there is a future time t_3)
$(\mathsf{out}_v?^c : (\mathsf{receive}, u, m)) \in tr_{t_3})$	(such that the message m will be delivered in tr_{t_3})

- Secondly, we have to specify the set of ports that should be scheduled by the fair master scheduler. For a given set $S_{\mathcal{H}}$, we define this set to be $\{\mathsf{net}^{\mathsf{rel}}_{u,v}{}^{\triangleleft}!, \mathsf{aut}^{\mathsf{rel}}_{u,v}{}^{\triangleleft}! \mid (u,v) \in \mathcal{I}_{\mathcal{H}}\}$.
- At last, the function t_s can be chosen arbitrary as long as it is bounded by a polynomial, i.e., $t_s(k) \in O(k^x)$ for a natural number x.

We can now state our main theorem.

Theorem 6.2 *The system $Sys^{\mathsf{rel}}_{\mathsf{id}}$ fulfills the polynomial liveness property $\varphi_{Sys^{\mathsf{rel}}_{\mathsf{id}}}$ perfectly, i.e., in formulas $Sys^{\mathsf{rel}}_{\mathsf{id}} \models_{\mathsf{perf}} \varphi_{Sys^{\mathsf{rel}}_{\mathsf{id}}}$.* □

Proof. Let an arbitrary structure $(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \in Sys^{\mathsf{rel}}_{\mathsf{id}}$, arbitrary polynomials J, t_{stop} with $t_{\mathsf{stop}} < t_s$ be given. We then define $Q_{\mathsf{X}}(k) := Q_{\mathsf{H}}(k) := t_{\mathsf{stop}}(k) + J(k) \cdot k^x$. Now, let $conf^{\mathsf{live}} = (\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}, \{\mathsf{H}\} \cup \{\mathsf{X}_{\mathsf{fair}}\}, \mathsf{A})^{\mathsf{live}} \in \mathsf{Conf}^{\mathsf{live}}(Sys^{\mathsf{rel}}_{\mathsf{id}})$ denote an arbitrary (t_{stop}, J, Q_{X}, Q_{H})-liveness configuration. We have to show that all $Q_{\mathsf{H}}(k)$ prefixes of the restriction of the run to the ports $S^{\mathsf{H}}_{\mathcal{H}}$ lie in $Req(S^{\mathsf{H}}_{\mathcal{H}})$.

In our particular case, this means the following: let an arbitrary pair $(u,v) \in \mathcal{I}_{\mathcal{H}}$ be given, and assume that the preconditions of $Req(S_{\mathcal{H}})$ are fulfilled, i.e., both users u and v have initialized themselves and a connection has been established between them at time l. Now, user u sends a command (send, m, v) at $\mathsf{TH}_{\mathcal{H}}$ at time $t > l$ and schedules it. Moreover, the message is valid and the machine of user u has not been stopped so far.

If we take a look at the "Send" transition of the machine $\mathsf{TH}_{\mathcal{H}}$, we can see that all of its internal checks will succeed by our above preconditions. Thus, it will output $(\mathsf{send\_blindly}, i, l, v)$ at $\mathsf{net}^{\mathsf{rel}}_{u,v}!$. By construction of $\mathsf{TH}_{\mathcal{H}}$ it only outputs anything at $\mathsf{net}^{\mathsf{rel}}_{u,v}!$ if it obtains inputs of the form $(\mathsf{send}, \cdot, v)$ at $\mathsf{in}_u?$. Since these inputs must come from the user u and the overall honest user H will stop outputting messages after its $t_{\mathsf{stop}}(k)$-th view-step, there can be at most $t_{\mathsf{stop}}(k)$ messages stored in the buffer $\widetilde{\mathsf{net}^{\mathsf{rel}}_{u,v}}$. By precondition, the function t_s (as a function of k) is bounded by k^x for a natural number x, so $t_{\mathsf{stop}}(k) < t_s(k)$ implies that the number of messages stored in $\widetilde{\mathsf{net}^{\mathsf{rel}}_{u,v}}$ is also bounded by k^x.

We now distinguish two cases: first, we assume that the user v is stopped before the overall user H stops outputting messages, i.e., before the run-empty phase begins. In this case, $\mathsf{TH}_{\mathcal{H}}$ will output a command (stop) at $\mathsf{out}_v!$ and schedule it by construction, so we have $(\mathsf{out}_v?^c : (\mathsf{stop})) \in tr_{t_2}$ for one particular $t_2 \in \mathbb{N}$. Hence, the requirement is fulfilled in this case, even before the run-empty phase begins.

Secondly, we assume that the machine of v has not been stopped before the run-empty phase begins. Since the adversary does not produce outputs any further, and $\mathsf{TH}_{\mathcal{H}}$ may only stop a machine after it has been scheduled by the adversary, the machine of v will not be stopped during the whole run. In this case, we have to prove that the message m will in fact be delivered.

As we already stated above, the desired message m (more precisely, the term $(\mathsf{send\_blindly}, i, l, v)$) has been stored in the buffer $\widetilde{\mathsf{net}^{\mathsf{rel}}_{u,v}}$. Moreover, this buffer can contain at most a polynomial number of messages (bounded by k^x). By precondition, the master scheduler $\mathsf{X}_{\mathsf{fair}}$ is J-fair, hence it schedules the first message of every of its connected buffers again and again, each one always after at most $J(k)$ steps. Since $\mathsf{net}^{\mathsf{rel}}_{u,v}{}^{\triangleleft}! \in \mathsf{ports}(\mathsf{X}_{\mathsf{fair}})$ holds by assumption, the buffer $\widetilde{\mathsf{net}^{\mathsf{rel}}_{u,v}}$ has to be scheduled after at most $J(k)$ view-steps of $\mathsf{X}_{\mathsf{fair}}$, so the term $(\mathsf{send\_blindly}, i, l, v)$ will be scheduled after at most $J(k) \cdot k^x$ view-steps of $\mathsf{X}_{\mathsf{fair}}$. If we now take a closer look at the behaviour of $\mathsf{TH}_{\mathcal{H}}$ in this case, we will see that all internal checks will succeed by assumption (the user v is initialized, a connection is established, v's machine has not been stopped, and the message m has been stored in $deliver^*_{u,v}[i]$ before). Thus, $\mathsf{TH}_{\mathcal{H}}$ outputs $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, so we have $(\mathsf{out}_v?^c : (\mathsf{receive}, u, m)) \in tr_{t_3}$ for one particular time $t_3 > t$. Note that our choice of Q_{X} and Q_{H} additionally ensures that both $\mathsf{X}_{\mathsf{fair}}$ and H run sufficiently long for this event to happen.

Now, we are almost finished. The only thing left to show is that this input occurs after a polynomial number of view-steps of the *user*; by now we only showed that it will happen after a polynomial number of view-steps of the *master scheduler*. However, since the user H does not produce any outputs any further, the master scheduler will always be scheduled immediately after the honest user. Thus, the number of view-steps the honest user can perform in the run-empty phase is bounded by the number of view-steps of $\mathsf{X}_{\mathsf{fair}}$. Therefore, the message will be received after at most $J(k) \cdot k^x$ view-steps of the honest users, counted from the beginning of the run-empty phase, i.e., after at most $t_{\mathsf{stop}}(k) + J(k) \cdot k^x = Q_{\mathsf{H}}$ view-steps in total, which finishes the proof. ∎

After proving the liveness property for the ideal specification, we now concentrate on the concrete implementation.

Theorem 6.3 *The real system $Sys^{\mathsf{rel}}_{\mathsf{real}}$ fulfills the polynomial liveness property $\varphi_{Sys^{\mathsf{rel}}_{\mathsf{real}}}$ computationally, with $\varphi_{Sys^{\mathsf{rel}}_{\mathsf{real}}}$ given as in Theorem 6.1. In formulas, $Sys^{\mathsf{rel}}_{\mathsf{real}} \models_{\mathsf{poly}} \varphi_{Sys^{\mathsf{rel}}_{\mathsf{real}}}$.* □

Proof. Obviously, perfect fulfillment of polynomial liveness implies fulfillment in the computational case. Thus, using Theorem 6.2, we know that $Sys^{\mathsf{id}}_{\mathsf{rel}} \models_{\mathsf{poly}} \varphi_{Sys^{\mathsf{id}}_{\mathsf{rel}}}$. As we already stated above, the concrete implementation is at least as secure as the abstract specification with respect to liveness in the computational case, i.e., $Sys^{\mathsf{rel}}_{\mathsf{real}} \geq^{f,\mathsf{live},\mathsf{poly}} Sys^{\mathsf{rel}}_{\mathsf{id}}$. Now the claim follows with Theorem 6.1. ∎

6.5 Conclusion

We have presented the first general definition of polynomial fairness and polynomial liveness in asynchronous reactive systems. We considered three grades of fulfilling a given polynomial liveness property: perfect (denoting usual fulfillment), statistical (denoting fulfillment up to a statistically small error probability) and computational (denoting fulfillment up to a negligible error probability, if all machines have polynomial runtime). Especially the computational case is essential to cope with real cryptography, since usually we can only ensure that good things happen if the underlying cryptographic primitives have not been broken, which might happen with negligible probability. Our approach might help to make the important concept of liveness better accessible for systems involving real cryptographic primitives. We have shown that polynomial liveness properties behave well under simulatability under certain conditions which enables step-wise refinement and modular proofs. Moreover, properties of abstract specifications can be validated by formal proof tools more easily than concrete implementations, although the polynomial-time limits t_s and J might make that more complicated for polynomial liveness than it is for safety properties. As an example fitting our definition, we have presented a specification of secure message transmission with reliable channels. Here, reliability is considered as the desired liveness property, and we have shown that the abstract specification in fact fulfills this property. Moreover, we have presented a concrete implementation, and, using our preservation theorem of the previous section, we have concluded that the implementation also fulfills this liveness property.

Chapter 7

Computational Probabilistic Non-Interference

After introducing integrity, liveness and fairness, we now finally concentrate on the important concept of non-interference, which has become very popular for expressing both integrity and privacy properties. We present the first general definition of probabilistic non-interference in reactive systems which includes a computational case. Similar to the previous chapter, this case is essential to cope with real cryptography since non-interference properties can usually only be guaranteed if the underlying cryptographic primitives have not been broken, which might happen, but with negligible probability. We show that our definition is maintained under simulatability, which allows secure composition of systems, and we present a general strategy how cryptographic primitives can be included in information flow proofs. As an example we present an abstract specification and a possible implementation of a cryptographic firewall guarding two honest users from their environment, and we prove them to fulfill our definition of non-interference. We conclude this chapter with a short summary of its results.

7.1 Introduction and Related Literature

Nowadays, information flow and non-interference are known as powerful possibilities for expressing privacy and integrity requirements a program or a cryptographic protocol should fulfill. Over the last two decades there has been considerable progress in this field of research which we will briefly review now. The first models for information flow have been considered for secure operating systems by Bell and LaPadula [8], and Denning [15]. After that, various models have been proposed that rigorously define when information flow is considered to occur. The first one was named *non-interference* introduced by Goguen and Meseguer [20, 21] in order to analyze the security of computer systems, but their work was limited to deterministic systems. Nevertheless, future work was (and still is) based on their idea of defining information flow. After that, research focused on non-deterministic systems mainly distinguishing between probabilistic and possibilistic behaviors. Beginning with Sutherland [60] the possibilistic case has been dealt with in [36, 64, 37, 66, 34], while security properties handling probabilistic and information-theoretic behaviours in non-deterministic systems have been proposed by Gray [23, 24] and McLean [38]. Clark et. al. showed in

[13] that absence of possibilistic information implies absence of probabilistic information flow.

Gray's definition of "Probabilistic Non-Interference" of reactive systems stands out. It is closely related to the perfect case of our definition, but it does not cover computational aspects of probabilistic non-interference which are essential for reasoning about systems using real cryptographic primitives. Thus, if we want to consider real cryptography we cannot only restrict ourselves to perfect non-interference as captured by the definition of Gray (nor to any other definition mentioned before, because they are non-probabilistic and hence not suited to cope with real cryptography) because it will not be sufficient for most cryptographic purposes. Adopting this notion we will present the first general definition of non-interference for this computational case, i.e., our definition can be used to reason about non-interference properties of arbitrary cryptographic primitives for the first time.

One important application of information flow is the static analysis of program code with respect to certain privacy requirements. This problem was first considered by Denning [16] by defining flow graphs on I/O variables. In recent times, type-based systems have been proposed [62, 58, 63, 57] for detecting and eliminating information flow in different kinds of languages. Some of these systems were proven correct by Sabelfeld and Sands [53, 54] by presenting a semantic characterization of probabilistic bisimulation that they used to express non-interference for multi-threaded and sequential programs. Mantel and Sabelfeld [35] investigated the integration of security properties of programming languages and abstract-level properties of information flow providing an interesting overview of how models of different security properties could be combined to increase the relative power of their analysis. Moreover, a tool for automatic checking of information flow in concurrent languages has been developed by Focardi and Gorrieri [19] for a variety of different information flow models.

Today, there is no general definition of non-interfered information flow but several of them coexist. Every definition has advantages and disadvantages so which one to take mainly depends on the goal to strive for. However, many definitions of non-interference have been overly restrictive preventing useful systems from being built. This problem is often tackled by downgrading certain information which then may nevertheless leak from the system, see [42, 67]. The amount of leaked information can in some cases be rigorously defined using information-theoretic techniques [40, 27].

Recent research also focused on expressing non-interference properties involving real cryptographic primitives. Laud [28] presented a sequential language where real computational secrecy can be expressed. Besides our work, this paper contains the only definition of non-interference including a computational case. However, only encryption is covered so far, i.e., other important concepts like authentication, pseudo-number generators, etc. are not considered. Moreover, the definition is non-reactive, i.e., it does not comprise continuous interaction between the user, the adversary, and the system, which is a severe restriction to the set of considered cryptographic systems. In contrast to that, our definition is reactive and comprises arbitrary cryptographic primitives. Volpano [61] investigated which conditions are needed so that one-way functions can be used safely in a programming language, but he did not actually express non-interference but only secrecy of a specific secret. Abadi and Blanchet [1] introduced type systems where asymmetric communication primitives, especially public-key encryption can be expressed, but these primitives are only relative to a Dolev-Yao abstraction [17], i.e., the primitives are idealized so that no computational non-interference definition is needed. For a discussion why the Dolev-Yao abstraction is not justified by current cryptography, see [48].

Our definition is closely related to the indistinguishability of probability distributions and it can be seen as a complementary approach to include cryptographic primitives in a definition of information flow. In contrast to most existing definitions we will not abstract from cryptographic details and probabilism, e.g., by using the common Dolev-Yao abstraction or special type systems, but we immediately include the computational variant in our definition. This enables sound reduction proofs with respect to the security definitions of the included cryptographic primitives (e.g., reduction proofs against the security of an underlying public key encryption scheme), i.e., a possibility to break the non-interference properties of the system can be used to break the underlying cryptography. Moreover, we show that our definition behaves well under the concept of simulatability modern cryptography often uses, i.e., non-interference properties proved for an abstract specification automatically carry over to the concrete implementation which enables modular designs and proofs. Thus, non-interference properties can be expressed for reactive systems containing arbitrary cryptographic primitives, which is of great importance for extensible systems like applets, kernel extensions, mobile agents, virtual privacy networks, etc. Exemplarily, we present a concrete implementation of a cryptographic firewall which enables two honest users to communicate with each other, but guards them from their environment, and we will prove the implementation to fulfill the desired non-interference property.

7.2 Expressing Non-Interference

In this section we define non-interference for our underlying model. At first we look at the more general topic of information flow. Information flow properties consist of two components: a *flow policy* and a *definition of information flow*. Before we turn our attention to the formal definition we start with an informal description of how these two components will be expressed. Flow policies are built by graphs with two different classes of edges. The first class symbolizes that information may flow between two users, the second class symbolizes that it may not. If we now want to define non-interference, we have to provide a semantics for the second class of edges. Intuitively, we want to express that there is no information flow from a user H_H to a user H_L iff the view of H_L does not change for every possible behaviour of H_H, i.e., H_L should not be able to distinguish arbitrary two families of views induced by two behaviours of H_H.

7.2.1 Flow Policies

We start by defining the flow policy graph.

Definition 7.1 *(General Flow Policy) A* general flow policy *is a pair $\mathcal{G} = (\mathcal{S}, \mathcal{E})$ with $\mathcal{E} \subseteq \mathcal{S} \times \mathcal{S} \times \{\rightsquigarrow, \not\rightsquigarrow\}$. Thus, we can speak of a graph $\mathcal{G}$ with two different kind of edges: $\rightsquigarrow, \not\rightsquigarrow \subseteq \mathcal{S} \times \mathcal{S}$. Furthermore we demand $(s_1, s_1) \in \rightsquigarrow$ for all $s_1 \in \mathcal{S}$, and every pair (s_1, s_2) of nodes should be linked by exactly one edge, so $\rightsquigarrow$ and $\not\rightsquigarrow$ form a partition of $\mathcal{S} \times \mathcal{S}$.* ◇

Remark. The set $\mathcal{S}$ often consists of only two elements $\mathcal{S} = \{L, H\}$ which are referred to as low- and high-level users. A typical flow policy would then be given by $L \rightsquigarrow L$, $L \rightsquigarrow H$, $H \rightsquigarrow H$, and finally $H \not\rightsquigarrow L$, see Figure 7.1, so there should not be any information flow from high- to low-level users.

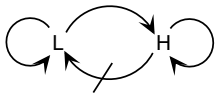

Figure 7.1: A Typical Flow Policy Graph Consisting of High and Low Users Only.

This definition is quite general since it uses an arbitrary set $\mathcal{S}$. If we want to use it for our purpose, we have to refine it so that it can be applied to a system Sys of our considered model. The intuition is to define a graph on the possible participants of the protocol, i.e., users and the adversary. However, this definition would depend on certain details of the users and the adversary, e.g., their port names, so we specify users by their corresponding specified ports of Sys, and the adversary by the remaining free ports of the system to achieve independence. After that, our flow policy only depends on the ports of Sys.

Definition 7.2 *(Flow Policy) Let a structure $(\hat{M}, S)$ be given, and let $\Gamma_{(\hat{M},S)} = \{S_i \mid i \in \mathcal{I}\}$ denote a partition of S for a finite index set $\mathcal{I}$, so $\Delta_{(\hat{M},S)} := \Gamma_{(\hat{M},S)} \cup \{\bar{S}\}$ is a partition of $\mathsf{free}([\hat{M}])$. A flow policy $\mathcal{G}_{(\hat{M},S)}$ of the structure $(\hat{M}, S)$ is now defined as a general flow policy $\mathcal{G}_{(\hat{M},S)} = (\Delta_{(\hat{M},S)}, \mathcal{E}_{(\hat{M},S)})$.*

The set of all flow policies for a structure $(\hat{M}, S)$ and a partition $\Delta_{(\hat{M},S)}$ of $\mathsf{free}([\hat{M}])$ will be denoted by $POL_{\hat{M},S,\Delta_{(\hat{M},S)}}$. Finally, a flow policy for a system Sys is a mapping

$$\phi_{Sys}: \quad Sys \to \bigcup_{(\hat{M},S)\in Sys} POL_{\hat{M},S,\Delta_{(\hat{M},S)}}$$
$$(\hat{M}, S) \mapsto \mathcal{G}_{(\hat{M},S)}$$

that assigns each structure $(\hat{M}, S)$ a flow policy $\mathcal{G}_{(\hat{M},S)} \in POL_{\hat{M},S,\Delta_{(\hat{M},S)}}$. ◇

We will simply write $\mathcal{G}$, Δ, and $\mathcal{E}$ instead of $\mathcal{G}_{(\hat{M},S)}$, $\Delta_{(\hat{M},S)}$, and $\mathcal{E}_{(\hat{M},S)}$ if the underlying structure is clear from the context. Additionally, we usually consider graphs with the following property: for blocks of ports $S_H, S_L \in \Delta$ with $(S_H, S_L) \in \not\rightsquigarrow$ there should not be a path from S_H to S_L consisting of "$\rightsquigarrow$"-edges only. We will refer to this property as *transitivity property* and speak of a *transitive flow policy*.

The relation $\not\rightsquigarrow$ is the non-interference relation of $\mathcal{G}$, so for two arbitrary blocks $S_H, S_L \in \Delta$, $(S_H, S_L) \in \not\rightsquigarrow$ means that no information flow must occur directed from the user connected to S_H to the user connected to S_L. The notion of a transitive flow policy is motivated by our intuition that if a user H_H should not be allowed to directly send any information to user H_L, he should also not be able to send information to H_L by involving additional users, and similarly for the adversary.

7.2.2 Definition of Non-Interference

We now have to define the semantics of our non-interference relation $\not\rightsquigarrow$. Usually, expressing this semantics is the most difficult part of the whole definition. In our underlying model, it is a little bit easier because we already have definitions for runs, views, and indistinguishability that can be used to express the desired semantics.

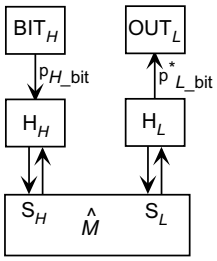

Figure 7.2: Sketch of our Non-Interference Definition

Figure 7.2 contains a sketch of our definition of non-interference between two users H_H and H_L (one of them might also be the adversary). Mainly, we define a specific machine BIT_H, which simply chooses a bit $b \in \{0,1\}$ at random and outputs it to H_H. Non-interference now means that H_H should not be able to change the view of H_L, so it should be impossible for H_L to output the bit b at $\mathsf{p}^*_{L\_\mathsf{bit}}!$ with a probability better than $\frac{1}{2}$ in the case of perfect non-interference. Statistical and computational non-interference now means that the advantage of H_L for a correct guess of b should be a function of a class $SMALL$ or negligible, respectively, measured in the given security parameter k.

These specific configurations including the special BIT_H- and OUT_L-machines will be called *non-interference configurations*. Note that these configurations are essentially based on the idea of guessing configurations (cf. Section 3.3). Before we turn our attention to the formal definition of these configurations we briefly describe which machines have to be included, how they behave, and which ports are essential for these sort of configurations, see Figure 7.3.

First of all, we have special machines BIT_H, OUT_L and $\mathsf{X}^{\mathsf{n\_in}}$. As described above, BIT_H will uniformly choose a bit at the start of the run and output it to the user H_H, while the second machine simply catches the messages received at port $\mathsf{p}^*_{L\_\mathsf{bit}}?$ to close the collection. This ensures that runs and views of the configuration are still defined without making any changes.

The machine $\mathsf{X}^{\mathsf{n\_in}}$ is the master scheduler of the configuration. Its function is to provide liveness and to avoid denial of service attacks, so it ensures that every machine will be able to send messages if it wants to. This is important for expressing non-interference, since a denial of service attack of the adversary will certainly be noticed by the affected user; thus, we have information flow. We now briefly describe the ports of the users. In order to improve readability we encourage the reader to compare these descriptions with the port labeling in Figure 7.3.

At first, we demand that every user i must not have any clockout ports; instead they have additional output ports $\mathsf{p}^{\mathsf{s}}!$ connected to the master scheduler for every specified clock-out port $\mathsf{p}^{\triangleleft}! \in S_i^c$, where S_i denotes the part of the partition user i connects to. The master scheduler will use these ports $\mathsf{p}^{\triangleleft}!$ to schedule outputs from port $\mathsf{p}!$, so a user can tell the master scheduler which port it wants to be scheduled. This is essential for excluding denial of service, because otherwise, there might be cycles inside of the system, so that neither the master scheduler nor some users will ever be scheduled. Therefore, we explicitly give the control to the master scheduler and define a useful scheduling strategy in the formal definition.

Secondly, we focus on the ports of the master scheduler. First of all, it has the corresponding input ports $\mathsf{p}^{\mathsf{s}}?$ for receiving scheduling demands from the users, and the corresponding clockout ports $\mathsf{p}^{\mathsf{s}\triangleleft}!$ and $\mathsf{p}^{\triangleleft}!$. Finally, it has special ports $\mathsf{master}_i!$ to

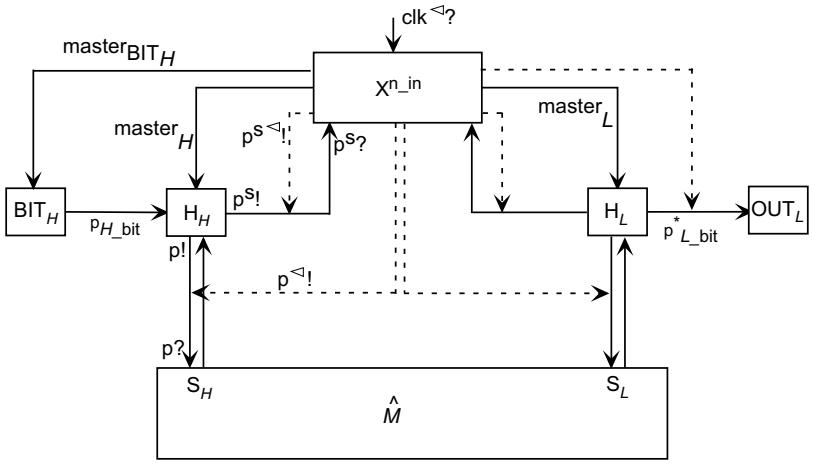

Figure 7.3: Sketch of a Non-Interference Configuration. The ports of the master scheduler $\mathsf{X}^{\mathsf{n\_in}}$ and of the two emphasized users H_H and H_L are sketched.

schedule (or to give control to) the user H_i. The actual scheduling process, i.e., how and in what order users are scheduled, is defined "round-robin like", i.e., we obtain a rotating clocking scheme such that every user and the adversary are clocked equally often with respect to the special master-ports. Let us turn to the formal definition.

Definition 7.3 *(Non-Interference Configuration)* Let a finite index set $\mathcal{I}$ with $\mathsf{A} \notin \mathcal{I}$ and $H, L \in \mathcal{I} \cup \{\mathsf{A}\}$, $H \neq L$ be given. Furthermore, let a multi-party configuration $conf^{\mathsf{mp}}_{H,L} = (\hat{M}, S, U \cup \{\mathsf{BIT}_H, \mathsf{OUT}_L, \mathsf{X}^{\mathsf{n\_in}}\}, \mathsf{A})$ of a system Sys with $U = \{\mathsf{H}_i \mid i \in \mathcal{I}\}$ and a partition $\Delta = \{S_i \mid i \in \mathcal{I}\} \cup \{\tilde{S}\}$ of $\mathsf{free}([\hat{M}])$ be given. For naming convention we will in the following set $\mathsf{H}_\mathsf{A} := \mathsf{A}$ and $S_\mathsf{A} := \tilde{S}$. We call this configuration a non-interference configuration for Δ of Sys if the following holds:

a) The ports of BIT_H are given by $\{\mathsf{master}_{\mathsf{BIT}_H}?, \mathsf{p}_{H\_\mathsf{bit}}!, \mathsf{p}_{H\_\mathsf{bit}}{}^{\triangleleft}!\}$.

The specific machine OUT_L has only one input port $\mathsf{p}^*_{L\_\mathsf{bit}}?$ connected to H_L.

The machine $\mathsf{X}^{\mathsf{n\_in}}$ is the master scheduler of the configuration. Its ports are given by

- $\{\mathsf{clk}^{\triangleleft}?\}$: The master clock-in port.
- $\{\mathsf{p}^{\triangleleft}! \mid \mathsf{p}^{\triangleleft}! \in S^c_i, i \in \mathcal{I}\}$: The ports for scheduling the buffers $\tilde{\mathsf{p}}$ with $\mathsf{p}^{\triangleleft}! \in S^c_i$.
- $\{\mathsf{p}^{\mathsf{s}}?, \mathsf{p}^{\mathsf{s}\triangleleft}! \mid \mathsf{p}^{\triangleleft}! \in S^c_i, i \in \mathcal{I}\}$: The ports connected to the users for receiving scheduling demands.
- $\{\mathsf{p}^{\mathsf{s}*}_{L\_\mathsf{bit}}?, \mathsf{p}^{\mathsf{s}*}_{L\_\mathsf{bit}}{}^{\triangleleft}!, \mathsf{p}^*_{L\_\mathsf{bit}}{}^{\triangleleft}!\}$: The ports for scheduling demands and outputs to machine OUT_L.
- $\{\mathsf{master}_i!, \mathsf{master}_i{}^{\triangleleft}! \mid i \in \mathcal{I} \cup \{\mathsf{A}\} \cup \{\mathsf{BIT}_H\}\}$: The ports for giving control to the users, the adversary and BIT_H.

b) For $i \in \mathcal{I}$ the ports of H_i must include $\{\mathsf{p}^{\mathsf{s}}! \mid \mathsf{p}^{\triangleleft}! \in S_i^c\} \cup \{\mathsf{master}_i?\}$, and the ports of the adversary must include $\mathsf{master}_\mathsf{A}?$. Additionally, H_H must have an input port $\mathsf{p}_{H\_\mathsf{bit}}?$, and H_L must have output ports $\mathsf{p}^*_{L\_\mathsf{bit}}!$ and $\mathsf{p}^{\mathsf{s}*}_{L\_\mathsf{bit}}!$ (cf. Figure 7.3).

Furthermore, we demand that the remaining ports of every user and of the adversary connect exactly to their intended subset of specified ports. Formally speaking, we demand that

$$\begin{aligned} & \mathsf{ports}(\mathsf{H}_H) \setminus (\{\mathsf{p}_{H\_\mathsf{bit}}?\} \cup \{\mathsf{master}_\mathsf{H}?\} \cup \{\mathsf{p}^{\mathsf{s}}! \mid \mathsf{p}^{\triangleleft}! \in S_H^c\}) \\ = \quad & S_H^c \setminus \{\mathsf{p}^{\triangleleft}! \mid \mathsf{p}^{\triangleleft}! \in S_H^c\} \end{aligned}$$

and

$$\begin{aligned} & \mathsf{ports}(\mathsf{H}_L) \setminus (\{\mathsf{p}^*_{L\_\mathsf{bit}}!, \mathsf{p}^{\mathsf{s}*}_{L\_\mathsf{bit}}!\} \cup \{\mathsf{master}_\mathsf{L}?\} \cup \{\mathsf{p}^{\mathsf{s}}! \mid \mathsf{p}^{\triangleleft}! \in S_L^c\}) \\ = \quad & S_L^c \setminus \{\mathsf{p}^{\triangleleft}! \mid \mathsf{p}^{\triangleleft}! \in S_L^c\} \end{aligned}$$

must hold, respectively.

For the remaining users H_i with $i \in \mathcal{I} \cup \{\mathsf{A}\}$, $i \notin \{H, L\}$ we simply have to leave out the special bit-ports, i.e., the equation

$$\mathsf{ports}(\mathsf{H}_i) \setminus (\{\mathsf{p}^{\mathsf{s}}! \mid \mathsf{p}^{\triangleleft}! \in S_i^c\} \cup \{\mathsf{master}_i?\}) = S_i^c \setminus \{\mathsf{p}^{\triangleleft}! \mid \mathsf{p}^{\triangleleft}! \in S_i^c\}$$

must hold.

If the adversary is one of the two emphasized users, i.e., $\mathsf{A} \in \{H, L\}$, we have to leave out the term $\{\mathsf{p}^{\mathsf{s}}! \mid \mathsf{p}^{\triangleleft}! \in S_H^c\}$, or $\{\mathsf{p}^{\mathsf{s}}! \mid \mathsf{p}^{\triangleleft}! \in S_L^c\}$ in the first two equalities, respectively.

c) The behaviour of the machine BIT_H is defined as follows. If BIT_H receives an arbitrary input at $\mathsf{master}_{\mathsf{BIT}_H}?$, it chooses a bit $b \in \{0, 1\}$ at random, outputs it at $\mathsf{p}_{H\_\mathsf{bit}}!$, and schedules it. The machine OUT_L simply does nothing on inputs at $\mathsf{p}^*_{L\_\mathsf{bit}}?$.

d) The behaviour of the machine $\mathsf{X}^{\mathsf{n\_in}}$ is defined as follows. Internally, it maintains two flags start and fl over $\{0, 1\}$, both initialized with 0, and a counter cnt over the finite index set $\mathcal{I} \cup \{\mathsf{A}\}$. Without loss of generality we assume $\mathcal{I} := \{1, \ldots, n\}$, so the counter is defined over $\{0, \ldots, n\}$, initialized with 0 (identifying the number 0 with A). Additionally, it has a counter $MaSc\_poly$ if the machine is demanded to be polynomial time, furthermore, a polynomial P must be given in this case that bounds the steps of $\mathsf{X}^{\mathsf{n\_in}}$.

If $\mathsf{X}^{\mathsf{n\_in}}$ is scheduled, it behaves as follows:

Case 1: Start of the run. If $\mathsf{start} = 0$: Set $\mathsf{start} := 1$ and output 1 at $\mathsf{master}_{\mathsf{BIT}_H}!$, 1 at $\mathsf{master}_{\mathsf{BIT}_H}{}^{\triangleleft}!$.

Case 2: Schedule users. If $\mathsf{fl} = 0$ and $\mathsf{start} = 1$: If $\mathsf{X}^{\mathsf{n\_in}}$ has to be polynomial time, it first checks $cnt = n$, increasing $MaSc\_poly$ in this case and checking whether $MaSc\_poly < P(k)$ holds for the security parameter k, stopping at failure. Now, it sets $cnt := cnt + 1 \bmod (n + 1)$, and outputs 1 at $\mathsf{master}_{cnt}!$, 1 at $\mathsf{master}_{cnt}{}^{\triangleleft}!$, i.e., it schedules the 'currently active' user H_{cnt} (or the adversary if $cnt = 0$). If $cnt \neq 0$, i.e., the clocked machine is an honest user, it additionally sets $\mathsf{fl} := 1$ to handle the scheduling demands of this user at its next clocking.

Case 3: Handling scheduling demands. If $\mathsf{fl} = 1$ and $\mathsf{start} = 1$: In this case it outputs 1 at every port $\mathsf{p}^{s\triangleleft}!$ with $\mathsf{p}^{\triangleleft}! \in S^c_{cnt}$ (for $cnt = L$ it also outputs 1 at $\mathsf{p}^{s*}_{L\_\mathsf{bit}}{}^{\triangleleft}!$) and tests whether it gets a non-empty input at exactly one input port.[1] If this does not hold, it sets $\mathsf{fl} := 0$ and does nothing. Otherwise, let $\mathsf{p}^s?$ denote the unique port with non-empty input i. $\mathsf{X}^{\mathsf{n\_in}}$ then outputs i at $\mathsf{p}^{\triangleleft}!$ and sets $\mathsf{fl} := 0$.

Non-interference configurations are denoted by $conf^{n\_in}_{H,L,\mathcal{I}} = (\hat{M}, S, U^{n\_in}, \mathsf{A})^{n\_in}_{\mathcal{I}}$ with $U^{n\_in} := U \cup \{\mathsf{BIT}_H, \mathsf{OUT}_L, \mathsf{X}^{\mathsf{n\_in}}\}$ but we will usually omit the index $\mathcal{I}$. $conf^{n\_in}_{H,L}$ is called polynomial-time if its underlying multi-party configuration $conf^{\mathsf{mp}}_{H,L}$ is polynomial-time. The set of all non-interference configurations of a system Sys for fixed H, L, and $\mathcal{I}$ will be denoted by $\mathsf{Conf}^{n\_in}_{H,L,\mathcal{I}}(Sys)$, and the set of all polynomial-time liveness configurations by $\mathsf{Conf}^{n\_in}_{H,L,\mathcal{I},\mathsf{poly}}(Sys)$. ◇

Definition 7.4 *(Non-Interference)* Let a flow policy $\mathcal{G} = (\Delta, \mathcal{E})$ for a structure $(\hat{M}, S)$ be given. Given two arbitrary elements $H, L \in \mathcal{I} \cup \{\mathsf{A}\}$, $H \neq L$ with $(S_H, S_L) \in \not\rightsquigarrow$, we say that $(\hat{M}, S)$ fulfills the non-interference requirement $NIReq_{H,L,\mathcal{G}}$

a) **perfectly** (written $(\hat{M}, S) \models_{\mathsf{perf}} NIReq_{H,L,\mathcal{G}}$) iff for any non-interference configuration $conf^{n\_in}_{H,L} \in \mathsf{Conf}^{n\_in}_{H,L,\mathcal{I}}(Sys)$ of this structure the inequality

$$P(b = b^* \mid r \leftarrow run_{conf^{n\_in}_{H,L},k}; b := r\lceil_{\mathsf{p}_{H\_\mathsf{bit}}!}; b^* := r\lceil_{\mathsf{p}^*_{L\_\mathsf{bit}}?}) \leq \frac{1}{2}$$

holds.

b) **statistically** for a class $SMALL$ ($(\hat{M}, S) \models_{SMALL} NIReq_{H,L,\mathcal{G}}$) iff for any non-interference configuration $conf^{n\_in}_{H,L} \in \mathsf{Conf}^{n\_in}_{H,L,\mathcal{I}}(Sys)$ of this structure there is a function $s \in SMALL$ such that that the inequality

$$P(b = b^* \mid r \leftarrow run_{conf^{n\_in}_{H,L},k}; b := r\lceil_{\mathsf{p}_{H\_\mathsf{bit}}!}; b^* := r\lceil_{\mathsf{p}^*_{L\_\mathsf{bit}}?}) \leq \frac{1}{2} + s(k)$$

holds. $SMALL$ must be closed under addition and with a function g also contain every function $g' \leq g$.

c) **computationally** ($(\hat{M}, S) \models_{\mathsf{poly}} NIReq_{H,L,\mathcal{G}}$) iff for any polynomial-time non-interference configuration $conf^{n\_in}_{H,L} \in \mathsf{Conf}^{n\_in}_{H,L,\mathcal{I},\mathsf{poly}}(Sys)$ the inequality

$$P(b = b^* \mid r \leftarrow run_{conf^{n\_in}_{H,L},k}; b := r\lceil_{\mathsf{p}_{H\_\mathsf{bit}}!}; b^* := r\lceil_{\mathsf{p}^*_{L\_\mathsf{bit}}?}) \leq \frac{1}{2} + \frac{1}{\mathsf{poly}(k)}$$

holds.

We write "$\models$" if we want to treat all cases together.

If a structure fulfills all non-interference requirements $NIReq_{H,L,\mathcal{G}}$ with $(S_H, S_L) \in \not\rightsquigarrow$, we say it fulfills the (global) requirement $NIReq_{\mathcal{G}}$ ($(\hat{M}, S) \models NIReq_{\mathcal{G}}$). A system Sys fulfills a flow policy ϕ_{Sys} if every structure $(\hat{M}, S) \in Sys$ fulfills its requirement $NIReq_{\phi_{Sys}(\hat{M},S)}$, and we consequently write $Sys \models NIReq_{\phi_{Sys}(\hat{M},S)}$, or $Sys \models \phi_{Sys}$ for short.

◇

[1] More formally, it enumerates these clockout ports and sends 1 at the first one. The buffer either schedules a message to $\mathsf{X}^{\mathsf{n\_in}}$ or it does nothing. In both cases $\mathsf{X}^{\mathsf{n\_in}}$ is scheduled again, so it can send 1 at the second clockout port and so on. Every received message is stored in an internal array so the test can easily be applied.

7.3 Preservation of Non-Interference Requirements under Simulatability

In this section we show that our definition of non-interference behaves well under simulatability. More precisely, we will show that the relation "at least as secure as" will not change the non-interference relation between two arbitrary users (one of them might also be the adversary), which yields the following preservation theorem. Similar to the properties introduced in the two precedent chapters, this theorem is essential for modular proofs and for the relation to formal proof tools.

Theorem 7.1 *(Preservation of Non-Interference Properties)* Let a flow policy ϕ_{Sys_2} for a system Sys_2 be given, so that $Sys_2 \models \phi_{Sys_2}$ holds. Furthermore, let a system Sys_1 be given with $Sys_1 \geq^f Sys_2$ for a valid mapping f. Then $Sys_1 \models \phi_{Sys_1}$ for all ϕ_{Sys_1} with $\phi_{Sys_1}(\hat{M}_1, S_1) := \phi_{Sys_2}(\hat{M}_2, S_2)$ for an arbitrary structure $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$. This holds for the perfect, statistical, and the computational case. □

Proof. We first show that ϕ_{Sys_1} is a well-defined flow policy for Sys_1 under our preconditions. Let an arbitrary structure $(\hat{M}_1, S_1) \in Sys_1$ be given. Simulatability implies that for every structure $(\hat{M}_1, S_1) \in Sys_1$, there exists $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$.

ϕ_{Sys_2} is a flow policy for Sys_2, so we have a flow policy $\mathcal{G}_{(\hat{M}_2, S_2)} = (\Delta, \mathcal{E})$ for $(\hat{M}_2, S_2)$. Furthermore, we have $S_1 = S_2$ by precondition, so we can indeed build the same set Γ of blocks on the specified ports and therefore the same partition Δ of the free ports of $[\hat{M}_1]$.[2] Hence, $\phi_{Sys_1}(\hat{M}_1, S_1)$ is defined on $(\hat{M}_1, S_1)$, so ϕ_{Sys_1} is a well-defined flow policy for Sys_1.

We now have show that Sys_1 fulfills ϕ_{Sys_1}. Let a structure $(\hat{M}_1, S_1) \in Sys_1$ and two elements $H, L \in \mathcal{I} \cup \{\mathsf{A}\}$, $H \neq L$ with $(S_H, S_L) \in \not\rightsquigarrow$ (with respect to the flow policy $\phi_{Sys_1}(\hat{M}_1, S_1)$) be given. We have to show that $(\hat{M}_1, S_1)$ fulfills the non-interference requirement $NIReq_{H,L,\mathcal{G}}$.

Let now a non-interference configuration $conf^{n\_in}_{H,L,1} = (\hat{M}_1, S_1, U^{n\_in}, \mathsf{A})^{n\_in} \in \mathsf{Conf}^{n\_in}_{H,L,\mathcal{I}}(Sys_1)$ be given. Because of $Sys_1 \geq^f Sys_2$ there is a configuration $conf_{H,L,2} = (\hat{M}_2, S_2, U^{n\_in}, \mathsf{A}') \in \mathsf{Conf}^{n\_in}_{H,L,\mathcal{I}}(Sys_2)$ for $(\hat{M}_2, S_2) \in f(\hat{M}_1, S_1)$ with $view_{conf^{n\_in}_{H,L,1}}(U^{n\_in}) \approx view_{conf_{H,L,2}}(U^{n\_in})$. Moreover, the honest users U^{n_in} are unchanged by simulatability, so $conf_{H,L,2}$ is again a non-interference configuration. Hence, we write $conf^{n\_in}_{H,L,2}$ in the following instead of $conf_{H,L,2}$. As usual we distinguish between the perfect, statistical, and the computational case. In the computational case, both configurations must be polynomial-time.

In the perfect case, we have $view_{conf^{n\_in}_{H,L,1}}(U^{n\_in}) = view_{conf^{n\_in}_{H,L,2}}(U^{n\_in})$ because of $Sys_1 \geq^f_{\mathsf{perf}} Sys_2$. Now, both $b := r\lceil_{\mathsf{p}_{H\_bit}!}$ and $b^* := r\lceil_{\mathsf{p}^*_{L\_bit}?}$ are part of the view of U^{n_in} because BIT_H and OUT_L are elements of U^{n_in}, so we obtain the same probabilities in both configurations. Our precondition $(\hat{M}_2, S_2) \models_{\mathsf{perf}} NIReq_{H,L,\mathcal{G}}$ and our arbitrary choice of $conf^{n\_in}_{H,L,1}$ implies that $(\hat{M}_1, S_1)$ also fulfills $NIReq_{H,L,\mathcal{G}}$.

We will treat the statistical and the computational case together. In the statistical (computational) case we have $view_{conf^{n\_in}_{H,L,1}}(U^{n\_in}) \approx_{SMALL} view_{conf^{n\_in}_{H,L,2}}(U^{n\_in})$ $(view_{conf^{n\_in}_{H,L,1}}(U^{n\_in}) \approx_{\mathsf{poly}} view_{conf^{n\_in}_{H,L,2}}(U^{n\_in}))$. We assume for contradiction

[2] More precisely, the block $\bar{S}_1$ is identified with $\bar{S}_2$. The ports of both sets may be different, but this does not matter because our definition of flow policies only uses whole blocks, so the different ports do not cause any trouble.

that $(\hat{M}_1, S_1)$ does not fulfill the non-interference requirement $NIReq_{H,L,\mathcal{G}}$, so the probability $p(k)$ of a correct guess for $b = b^*$ is not smaller than $\frac{1}{2} + s(k)$ for all $s \in SMALL$ in the statistical case (or $p(k) - \frac{1}{2}$ is not negligible in the computational case). Thus, the advantage $\epsilon(k) := p(k) - \frac{1}{2}$ of the adversary is not contained in $SMALL$ (or $\epsilon(k)$ is not negligible). $(\hat{M}_2, S_2)$ fulfills the non-interference requirement, so in this configuration, the advantage $\epsilon'(k)$ for a correct guess is a function of $SMALL$ in the statistical or negligible in the computational case.

We can then define a distinguisher Dis as follows. Given the views of U^{n_in} in both configurations it explicitly knows the views of BIT_H and OUT_L. Now Dist outputs 1 if $b = b^*$ and 0 otherwise. Its advantage in distinguishing is

$$\begin{aligned} & |P(\mathsf{Dis}(1^k, view_{conf^{n\_in}_{H,L,1},k}(U^{n\_in})) = 1) \\ & -P(\mathsf{Dis}(1^k, view_{conf^{n\_in}_{H,L,2},k}(U^{n\_in})) = 1)| \\ = \; & |\frac{1}{2} + \epsilon(k) - (\frac{1}{2} + \epsilon'(k))| \\ = \; & \epsilon(k) - \epsilon'(k). \end{aligned}$$

For the polynomial case, this immediately contradicts our assumption $Sys_1 \geq^f_{\mathsf{poly}} Sys_2$ because $\epsilon(k) - \epsilon'(k)$ is not negligible. For the statistical case, the distinguisher Dis can be seen as a function on the random variables, so Lemma 2.4 implies

$$\begin{aligned} & \Delta(view_{conf^{n\_in}_{H,L,1},k}(U^{n\_in}), view_{conf^{n\_in}_{H,L,2},k}(U^{n\_in})) \\ \geq \; & |P(\mathsf{Dist}(1^k, view_{conf^{n\_in}_{H,L,1},k}(U^{n\_in})) = 1) \\ & -P(\mathsf{Dist}(1^k, view_{conf^{n\_in}_{H,L,2},k}(U^{n\_in})) = 1)| \\ = \; & \epsilon(k) - \epsilon'(k). \end{aligned}$$

But $\epsilon(k) - \epsilon'(k) \notin SMALL$ must hold, because $\epsilon'(k) \in SMALL$, and we demanded the class $SMALL$ to be closed under addition. Thus, we have $\Delta(view_{conf^{n\_in}_{H,L,1},k}(U^{n\_in}), view_{conf^{n\_in}_{H,L,2},k}(U^{n\_in})) \notin SMALL$ because $SMALL$ is closed under making functions smaller which yields the desired contradiction. ■

7.4 A Cryptographic Firewall

In the following we present an example of a system that allows authenticated communication between two users and furthermore ensures that these two users cannot be affected by their environment. This yields a flow policy our system has to (and indeed will) fulfill.

The construction of both our ideal and our real system can be explained using Figure 2.3 on page 20. Our ideal specification is based on the ideal specification for secure message transmission with perfectly ordered channels, introduced in chapter 4 of this work, which we will slightly modify to fit our requests. Mainly, we have to avoid denial of service attacks. We will denote this modified ideal system by Sys'_0 following the notation of Figure 2.3 on page 20. Furthermore, a possible implementation has also been presented in chapter 4 which we modify in the same way as the ideal specification to maintain the *at least as secure as* relation.

Our cryptographic firewall will then be derived by defining a new system Sys_1 so that combination with Sys'_0 yields the ideal system, replacing Sys'_0 with Sys_0 finally

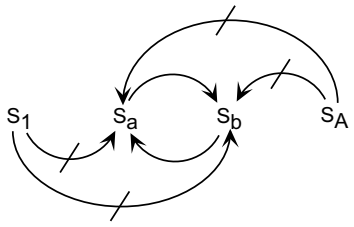

Figure 7.4: Sketch of the flow policy $\mathcal{G}$ of our system. Only one non-emphasized user S_1 is considered and some edges of the graph are omitted. Missing edges are of the form "$\rightsquigarrow$".

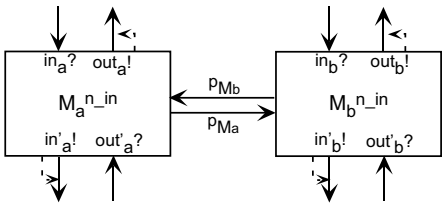

Figure 7.5: Sketch of system Sys_1.

yields a possible implementation. Sys_1 will be designed to filter messages sent by "wrong" senders that should not be allowed to influence the special users according to the flow policy shown in Figure 7.4. According to Figure 2.3, we denote our ideal system as Sys^* and our real implementation as $Sys^\#$.

At first, we show how to modify the ideal system for secure message transmission with ordered channels such that non-interference can be achieved. After that, we introduce our system Sys_1 and prove that our ideal system Sys^* fulfills its non-interference requirements. The last step we have to do is deriving a real system that fulfills its non-interference requirements. We already stated that $Sys_0 \geq Sys'_0$ holds for the unmodified systems. We will briefly sketch that it also holds for the modified systems, so $Sys^\#$ is at least as secure as Sys^* by the composition theorem. Now Theorem 7.1 implies that the real system $Sys^\#$ also fulfills its requirements which successfully finishes our attempt to design a real example that fits our non-interference definition.

7.4.1 The Ideal System

Let n denote the number of participants, $\mathcal{I} := \{1, \ldots, n\}$ the set of indices of the considered participants, and $\mathcal{I}_\mathsf{A} := \mathcal{I} \cup \{\mathsf{A}\}$ the set of participants including the adversary. In the following we will identify these indices with their corresponding user. Intuitively, we want a system that fulfills the flow policy shown in Figure 7.4. We consider two distinguished users a and b with $\{a, b\} \in \mathcal{I}$. We now have two blocks of specified ports S_a and S_b, so that information must not flow to one of these blocks from the outside. More precisely, we have non-interference requirements $NIReq_{i_1,i_2,\mathcal{G}}$ for every pair (i_1, i_2) with $i_1 \in \mathcal{I}_\mathsf{A} \setminus \{a, b\}$, $i_2 \in \{a, b\}$.

Recall, that the specification of secure message transmission with ordered channels is of the typical form

$$Sys'_0 = \{(\{\mathsf{TH}_\mathcal{H}\}, S_\mathcal{H}) | \mathcal{H} \subseteq \mathcal{M}\},$$

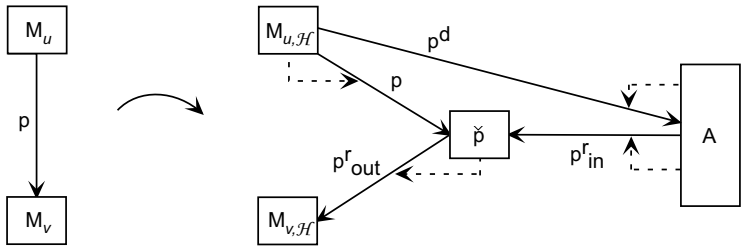

Figure 7.6: Self-scheduled reliable, non-authenticated channels

i.e., there is one structure for every subset of the machines, denoting the honest users.

Necessary Modifications of the scheme. We want our system to fulfill our flow policy shown in Figure 7.4, so especially the non-interference requirement $NIReq_{A,a,\mathcal{G}}$ must hold. If we explicitly allow the adversary to schedule the communication between H_a and H_b he can obviously achieve two distinguishable behaviours by denial of service attacks as follows. On the one hand, he directly schedules every message sent from H_b to H_a in one behaviour, on the other hand he does not schedule any message sent from H_b to H_a. This problem cannot be solved by the filtering system Sys_1 if scheduling of the communication channel is done by the adversary.[3] In practice, this means that two persons will not be able to communicate without being interfered from outside if the channel they use can be cut off by the adversary. A possible solution for the problem is to use reliable authenticated channels for key exchange and reliable, non-authenticated channels for sending of messages (cf. Definition 6.6 and Scheme 6.1) between the two emphasized users a and b. Obviously, channels which are reliable *and* authenticated could be used as well for sending of messages, but in this case, we would no longer need the underlying cryptography (e.g., authentication), cf. the precedent chapter. Therefore, we only consider these authenticated channels for key exchange as usual, but sending of messages is still performed over non-authenticated channels.

However, we simplify definition 6.6 of reliable, non-authenticated channels according to Figure 7.6. Just as in the original definition the output of the machine M_u at a port p is duplicated at a port p^{d} as usual, but the machine M_u now has the corresponding clockout port $\mathsf{p}^{\triangleleft}!$ and immediately schedules this output. This corresponds to a still non-authenticated channel, but reliability is now guaranteed by immediately delivering the message to its recipient. The reasons for this modification are quite simple: At first, it is not immediately clear how the results of the precedent chapter could be applied, i.e., we could assume the master scheduler $\mathsf{X}^{\mathsf{n\_in}}$ to schedule the port $\mathsf{p}^{\triangleleft}!$. However, this would imply $\mathsf{p}^{\triangleleft}! \in S$, so we have to include it in the considered partition of the flow policy, which would unnecessarily complicate the whole system, and its proof. Second, the system should serve as an illustration of our definition of non-interference, so it should be as easy as possible without containing unnecessary accessory parts.

Similarly, we consider self-scheduled authenticated channels for key exchange, so the keys are sent to the adversary as usual, but also immediately scheduled to the recipient.

[3] Sys_1 can only sort out messages from "wrong" senders, the messages mentioned are sent by the "valid" user H_b, so they have to be delivered because Sys_1 has no internal clock to check for denial of service attacks.

These modifications carry over to the trusted host $\mathsf{TH}_{\mathcal{H}}$ as follows:

- If $\mathsf{TH}_{\mathcal{H}}$ receives an input $(\mathsf{snd\_init})$ from H_a, it implicitly initializes a communication with H_b and outputs $(\mathsf{snd\_init})$ to the adversary, $(\mathsf{rec\_init}, a)$ to H_b and schedules the second output.
- If $\mathsf{TH}_{\mathcal{H}}$ receives an input (send, m, b) from H_a, it outputs $(\mathsf{send\_blindly}, i, l, b)$ to the adversary, $(\mathsf{receive}, m, a)$ to H_b scheduling the second output.

These modifications are also done for switched variables a and b. Strictly speaking, we obtain the following scheme.

Scheme 7.1 Let $n \in \mathbb{N}$ and polynomials $L, s_1, s_2 \in \mathbb{N}[x]$ be given that bound the length of each message and the number of messages a user can send of receive, respectively. Let $\mathcal{M} := \{1, \ldots, n\}$ denote the set of possible participants and the access structure $\mathcal{ACC}$ is the set of all subsets $\mathcal{H}$ of $\mathcal{M}$ including the elements a and b, i.e., $\{a, b\} \subseteq \mathcal{H}$ must hold. Our specification for secure message transmission with (simplified) reliable, ordered channels is now a standard ideal system

$$Sys'_0 = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \{a, b\} \subseteq \mathcal{H} \subseteq \mathcal{M}\}.$$

with the standard localized definition $S^c_{\mathcal{H}} := \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$, and $\mathsf{TH}_{\mathcal{H}}$ defined as follows. When $\mathcal{H}$ is clear from the context, let $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$ denote the indices of corrupted machines.

The ports of the machine $\mathsf{TH}_{\mathcal{H}}$ are given by $\{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\} \cup \{\mathsf{from\_adv}_u?, \mathsf{to\_adv}_u!, \mathsf{to\_adv}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$.

Internally, $\mathsf{TH}_{\mathcal{H}}$ maintains seven arrays: it contains an array $(init^{\mathsf{spec}}_{u,v})_{u,v\in\mathcal{M}}$ over $\{0,1\}$ that models initialization of users, an array $(sc\_in^{\mathsf{spec}}_{u,v})_{u\in\mathcal{H},v\in\mathcal{M}}$ over $\{0, \ldots, s_1(k)\}$ counting the number of outgoing messages of user u intended to user v, arrays $(msg\_out^{\mathsf{spec}}_{u,v})_{u,v\in\mathcal{H}}$, $(sc\_out^{\mathsf{spec}}_{u,v})_{u\in\mathcal{M},v\in\mathcal{H}}$ over $\{0, \ldots, s_2(k)\}$, an array $(stopped^{\mathsf{spec}}_u)_{u\in\mathcal{H}}$ over $\{0,1\}$, and an array $(msg\_in^{\mathsf{spec}}_{u,v})_{u\in\mathcal{H},v\in\mathcal{M}}$ over $\{0, \ldots, s_1(k)\}$. All six arrays should be initialized with 0 everywhere, except for $msg\_out^{\mathsf{spec}}_{u,v}$ being initialized with 1 everywhere. Finally, it contains an array $(deliver^{\mathsf{spec}}_{u,v})_{u,v\in\mathcal{H}}$ of lists, all initially empty. The state-transition function of $\mathsf{TH}_{\mathcal{H}}$ is defined by the following rules.

Initialization.

- **Send Initialization.** On input $(\mathsf{snd\_init})$ at $\mathsf{in}_u?$: If $sc\_in^{\mathsf{spec}}_{u,v} < s_1(k)$ for all $v \in \mathcal{M}$, set $sc\_in^{\mathsf{spec}}_{u,v} := sc\_in^{\mathsf{spec}}_{u,v} + 1$ for all $v \in \mathcal{M}$, otherwise do nothing. If the test holds check $stopped^{\mathsf{spec}}_u = 0$ and $init^{\mathsf{spec}}_{u,u} = 0$. In this case set $init^{\mathsf{spec}}_{u,u} := 1$. If $u = a$ $(u = b)$ set $init^{\mathsf{spec}}_{a,b} = 1$ $(init^{\mathsf{spec}}_{b,a} = 1)$ and output $(\mathsf{snd\_init})$ at $\mathsf{to\_adv}_u!$, $(\mathsf{rec\_init}, a)$ at $\mathsf{out}_b!$ and 1 at $\mathsf{out}_b{}^{\triangleleft}!$ ($(\mathsf{snd\_init})$ at $\mathsf{to\_adv}_u!$, $(\mathsf{rec\_init}, b)$ at $\mathsf{out}_a!$ and 1 at $\mathsf{out}_a{}^{\triangleleft}!$). Otherwise output $(\mathsf{snd\_init})$ at $\mathsf{to\_adv}_u!$ and 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$.
- **Receive initialization.** On input $(\mathsf{rec\_init}, u)$ at $\mathsf{from\_adv}_v?$ with $u \in \mathcal{M}, v \in \mathcal{H}$: If $stopped^{\mathsf{spec}}_v = 0$, $init^{\mathsf{spec}}_{u,v} = 0$, and $[u \in \mathcal{H} \Rightarrow init^{\mathsf{spec}}_{u,u} = 1]$, set $init^{\mathsf{spec}}_{u,v} := 1$. If $sc\_out^{\mathsf{spec}}_{u,v} < s_2(k)$ set $sc\_out^{\mathsf{spec}}_{u,v} := sc\_out^{\mathsf{spec}}_{u,v} + 1$, output $(\mathsf{rec\_init}, u)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

Sending and receiving messages.

- **Send.** On input (send, m, v) at $\mathsf{in}_u?$: If $sc\_in^{\mathsf{spec}}_{u,v} < s_1(k)$ and $stopped^{\mathsf{spec}}_u = 0$, set $sc\_in^{\mathsf{spec}}_{u,v} := sc\_in^{\mathsf{spec}}_{u,v} + 1$, otherwise do nothing. If $m \in \Sigma^+, l := \mathsf{len}(m) \leq L(k), v \in \mathcal{M} \setminus \{u\}$, $init^{\mathsf{spec}}_{u,u} = 1$ and $init^{\mathsf{spec}}_{v,u} = 1$ holds:

  If $v \in \mathcal{A}$ then { set $msg\_in^{\mathsf{spec}}_{u,v} := msg\_in^{\mathsf{spec}}_{u,v} + 1$ and output $(\mathsf{send}, (m, msg\_in^{\mathsf{spec}}_{u,v}), v)$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$ } else {set $i := \mathsf{size}(deliver^{\mathsf{spec}}_{u,v}) + 1$, $msg\_in^{\mathsf{spec}}_{u,v} := msg\_in^{\mathsf{spec}}_{u,v} + 1$, $deliver^{\mathsf{spec}}_{u,v}[i] := (m, msg\_in^{\mathsf{spec}}_{u,v})$.

  If $u \notin \{a, b\}$ or $v \notin \{a, b\}$ output $(\mathsf{send\_blindy}, i, l, v)$ at $\mathsf{to\_adv}_u!$, 1 at $\mathsf{to\_adv}_u{}^{\triangleleft}!$, if $\{u, v\} = \{a, b\}$ with $u \neq v$ set $msg\_out^{\mathsf{spec}}_{u,v} := msg\_in^{\mathsf{spec}}_{u,v} + 1$ and output $(\mathsf{send\_blindy}, i, l, v)$ at $\mathsf{to\_adv}_u!$, $(\mathsf{receive}, m, u)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$ [4]}.

- **Receive from honest party u.** On input $(\mathsf{receive\_blindly}, u, i)$ at $\mathsf{from\_adv}_v?$ with $u, v \in \mathcal{H}$: If $stopped^{\mathsf{spec}}_v = 0$, $init^{\mathsf{spec}}_{v,v} = 1$, $init^{\mathsf{spec}}_{u,v} = 1$, $sc\_out^{\mathsf{spec}}_{u,v} < s_2(k)$ and $(m, j) := deliver^{\mathsf{spec}}_{u,v}[i] \neq \downarrow$, check $msg\_out^{\mathsf{spec}}_{u,v} = j$. If this holds set $sc\_out^{\mathsf{spec}}_{u,v} := sc\_out^{\mathsf{spec}}_{u,v} + 1$, $msg\_out^{\mathsf{spec}}_{u,v} := j + 1$ and output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

- **Receive from dishonest party u:** On input $(\mathsf{receive}, u, m)$ at $\mathsf{from\_adv}_v?$ with $u \in \mathcal{A}, m \in \Sigma^+, \mathsf{len}(m) \leq L(k)$ and $v \in \mathcal{H}$: If $stopped^{\mathsf{spec}}_v = 0$, $init^{\mathsf{spec}}_{v,v} = 1$, $init^{\mathsf{spec}}_{u,v} = 1$ and $sc\_out^{\mathsf{spec}}_{u,v} < s_2(k)$, set $sc\_out^{\mathsf{spec}}_{u,v} := sc\_out^{\mathsf{spec}}_{u,v} + 1$ and output $(\mathsf{receive}, u, m)$ at $\mathsf{out}_v!$, 1 at $\mathsf{out}_v{}^{\triangleleft}!$.

- **Stop.** On input (stop) at $\mathsf{from\_adv}_u?$ with $u \in \mathcal{H}$: If $stopped^{\mathsf{spec}}_u = 0$, set $stopped^{\mathsf{spec}}_u := 1$ and output (stop) at $\mathsf{out}_u!$, 1 at $\mathsf{out}_u{}^{\triangleleft}!$.

$\diamond$

Putting it all together $\mathsf{TH}_{\mathcal{H}}$ has been modified as follows. First of all, it behaves identically for users $u \notin \{a, b\}$. In the following we consider a user $u \in \{a, b\}$ and write $\bar{u}$ for the corresponding emphasized user, i.e. $\bar{u} \in (\{a, b\} \setminus \{u\})$. If u sends its initialization, the new $\mathsf{TH}_{\mathcal{H}}$ immediately initializes a communication with $\bar{u}$. In the real system this corresponds to usual initialization but scheduling is done by the machine of the user, not by the adversary. Additionally, an initialization command is sent to the adversary corresponding to a self-scheduled authenticated channel. Note that the adversary will not be able to influence both u and $\bar{u}$ in the initialization phase by construction of $\mathsf{TH}_{\mathcal{H}}$ (additional initialization commands sent by the adversary will always be sorted out by $\mathsf{TH}_{\mathcal{H}}$). Furthermore, sending of messages between u and $\bar{u}$ has been changed. If u sends a message to $\bar{u}$, $\mathsf{TH}_{\mathcal{H}}$ directly outputs this message at port $\mathsf{out}_{\bar{u}}!$ and schedules it, additionally it outputs the usual blinded term to the adversary.

After the modification of Sys'_0 we can turn our attention to the system Sys_1. After that, we will show that the already mentioned simplified reliable channels can be used to modify the real system Sys_0 such that the relation $Sys_0 \geq Sys'_0$ still holds for the modified systems. The system Sys_1 is built by additional machines $\mathsf{M}^{\mathsf{n\_in}}_u$ for $u \in \{a, b\}$. These machines will be inserted between the users and the trusted host $\mathsf{TH}_{\mathcal{H}}$, see Figure 7.7. Formally, we obtain the following scheme:

[4] Increasing the message counter is essential for avoiding replay attacks because the message m is directly delivered to v using a reliable channel.

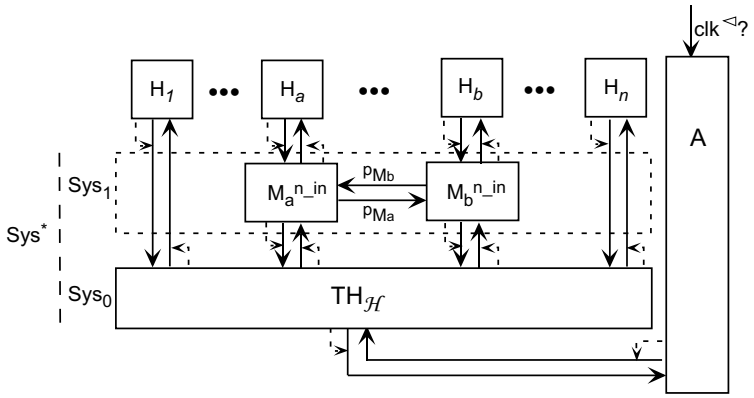

Figure 7.7: Ideal System for Non-Interfered Communication.

Scheme 7.2 (Sys_1) Let $n \in \mathbb{N}$ and polynomials $L, s, s' \in \mathbb{N}[x]$ be given. Here n denotes the number of intended participants, $L(k)$ bounds the message length and $s(k), s'(k)$ bound the number of messages each user can send and receive, respectively, for the security parameter k. Let $\mathcal{I} := \{1, \ldots, n\}$ denote the set of possible users again and $a, b \in \mathcal{I}$ the special users that should not be influenced from outside. The system Sys_1 is then defined as

$$Sys_1 := \{(\hat{M}, S)\}$$

with $\hat{M} = \{\mathsf{M}_a^{\mathsf{n\_in}}, \mathsf{M}_b^{\mathsf{n\_in}}\}$. The specified ports S are given by $S^c := \{\mathsf{out}_u?, \mathsf{in}_u!, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \{a, b\}\} \cup \{\mathsf{in}'_u?, \mathsf{out}'_u!, \mathsf{out}'_u{}^{\triangleleft}! \mid u \in \{a, b\}\}$. Without loss of generality we just describe the ports and the behaviour of machine $\mathsf{M}_a^{\mathsf{n\_in}}$. The machine $\mathsf{M}_b^{\mathsf{n\_in}}$ is defined analogously by exchanging the variables a and b. The ports of machine $\mathsf{M}_a^{\mathsf{n\_in}}$ are $\{\mathsf{in}_a?, \mathsf{out}_a!, \mathsf{out}_a{}^{\triangleleft}!\} \cup \{\mathsf{out}'_a?, \mathsf{in}'_a!, \mathsf{in}'_a{}^{\triangleleft}!\} \cup \{\mathsf{p}_{\mathsf{M}_b}?, \mathsf{p}_{\mathsf{M}_a}!, \mathsf{p}_{\mathsf{M}_a}{}^{\triangleleft}!\}$. The resulting connection graph is shown is Figure 7.5.

Internally, $\mathsf{M}_a^{\mathsf{n\_in}}$ maintains a counter $s_a \in \{0, \ldots, s(k)\}$ and an array $(s'_{a,u})_{u \in \mathcal{I}}$ over $\{0, \ldots, s'(k)\}$ bounding the number of messages H_a can send and receive, respectively, and a variable $stopped_a \in \{0, 1\}$ both initialized with 0 everywhere. The state-transition function of $\mathsf{M}_a^{\mathsf{n\_in}}$ is defined by the following rules.

Initialization.

- **Send initialization.** On input $(\mathsf{snd\_init})$ at $\mathsf{in}_a?$: If $s_a < s(k)$ it sets $s_a := s_a + 1$, otherwise it stops. If $stopped_a = 0$ it outputs $(\mathsf{snd\_init})$ at $\mathsf{in}'_a!$, 1 at $\mathsf{in}'_a{}^{\triangleleft}!$, otherwise it outputs $(\mathsf{snd\_init})$ at $\mathsf{p}_{\mathsf{M}_a}!$, 1 at $\mathsf{p}_{\mathsf{M}_a}{}^{\triangleleft}!$.

- **Receive initialization.** On input $(\mathsf{rec\_init}, u)$ at $\mathsf{out}'_a?$: It first checks whether $s'_{a,u} < s'(k)$ hold. In this case it sets $s'_{a,u} = s'_{a,u} + 1$, otherwise it stops. If $stopped_a = 0$ it checks $u = b$. If this also holds it outputs $(\mathsf{rec\_init}, b)$ at $\mathsf{out}_a!$ and 1 at $\mathsf{out}_a{}^{\triangleleft}!$. On input $(\mathsf{rec\_init}, b)$ at $\mathsf{p}_{\mathsf{M}_b}?$, it outputs $(\mathsf{rec\_init}, b)$ at $\mathsf{out}_a!$ and 1 at $\mathsf{out}_a{}^{\triangleleft}!$.

Sending and receiving messages.

- **Send.** On input (send, m, v) at $\mathsf{in}_a?$ with $m \in \Sigma^+$ and $\mathsf{len}(m) \leq L(k)$ it checks whether $s_a < s(k)$. If this holds it sets $s_a := s_a + 1$, otherwise it stops. If

$stopped_a = 0$ holds, it outputs (send, m, v) at $\mathsf{in}'_a!$, 1 at $\mathsf{in}'^{\triangleleft}_a!$. Otherwise it first checks $v = b$. After a successful test it outputs $(\mathsf{receive}, a, m)$ at $\mathsf{p}_{\mathsf{M}_a}!$ and 1 at $\mathsf{p}^{\triangleleft}_{\mathsf{M}_a}!$.

- **Receive.** On input $(\mathsf{receive}, u, m)$ at $\mathsf{out}'_a?$ it first checks whether $s'_{a,u} < s'(k)$. If this holds it sets $s'_{a,u} := s'_{a,u} + 1$, otherwise it stops. If $u = b$ holds it outputs $(\mathsf{receive}, b, m)$ at $\mathsf{out}_a!$ and 1 at $\mathsf{out}^{\triangleleft}_a!$. On input $(\mathsf{receive}, b, m)$ at $\mathsf{p}_{\mathsf{M}_b}?$ it outputs $(\mathsf{receive}, b, m)$ at $\mathsf{out}_a!$ and 1 at $\mathsf{out}^{\triangleleft}_a!$.

- **Stop.** On input (stop) at $\mathsf{out}'_a?$ or $\mathsf{p}_{\mathsf{M}_b}?$: If $stopped_a = 0$, it sets $stopped_a = 1$ and outputs (stop) at $\mathsf{p}_{\mathsf{M}_a}!$ and 1 at $\mathsf{p}^{\triangleleft}_{\mathsf{M}_a}!$.

$\diamond$

The special communication ports $\mathsf{p}_{\mathsf{M}_a}$ and $\mathsf{p}_{\mathsf{M}_b}$ are just included to prevent denial of service attacks. Recall, that a mighty attacker could simply overpower the machine of an honest user by sending too many messages, i.e., to exceed its runtime bound in the real world. In the ideal system this is modeled by letting the adversary stop arbitrary machines any time he likes. If we now consider an adversary that stops the machine of user a at the very start of the run and another one that never stops this machine, we would certainly obtain different views for this user. This problem cannot really be avoided if we do not provide additional channels for communication that guarantee availability. In practice this would correspond to a connection that contains trash all the time sent by the adversary, so the users (their machines in our case) would certainly look for a new way to communicate. Furthermore, this problem is much weaker in practice than in theory because it ought to be impossible (or at least very difficult) for an adversary to overpower a machine (the machine would surely be able to take countermeasures). If we would not consider these sorts of attacks the ports $\mathsf{p}_{\mathsf{M}_a}$ and $\mathsf{p}_{\mathsf{M}_b}$ could as well be omitted. Finally, a stopped machine $\mathsf{M}_a^{\text{n-in}}$ would want the machine $\mathsf{M}_b^{\text{n-in}}$ also to use the special communication ports, so it will 'stop' the machine as soon it has been stopped itself. Before we now build the combination of both systems to obtain our complete system Sys^*, we rename the ports $\mathsf{in}_u?$, $\mathsf{out}_u!$ and $\mathsf{out}^{\triangleleft}_u!$ of Sys'_0 into $\mathsf{in}'_u?$, $\mathsf{out}'_u!$ and $\mathsf{out}'^{\triangleleft}_u!$, respectively, for $u \in \{a, b\}$. Furthermore, we restrict the structures of Sys'_0 to all sets $\mathcal{H}$ with $\{a, b\} \subseteq \mathcal{H}$. Combination now means that we combine every structure of Sys'_0 with the (only) structure of Sys_1. The resulting system $Sys^* = \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \mid \{a, b\} \subseteq \mathcal{H} \subseteq \mathcal{I}\}$ is shown in Figure 7.7.

Remark 7.1. It is quite obvious how to modify the system Sys_1 to an arbitrary set of users (instead of $\{a, b\}$) that have to be guarded by the firewall. Moreover, we can easily consider multiple disjoint sets of users so that a user can communicate with other users of its own set without being interfered from outside. This corresponds to multiple firewalls and can easily be achieved by modifying the filtering system Sys_1, so our specification carries over to arbitrary transitive flow policies. $\circ$

7.4.2 The Real System

We now briefly sketch how to modify the real system for secure message transmission with ordered channels such that the relation "at least as secure as" is preserved.

Obviously, the modification in the "Send initialization" transition can be modeled similar by letting the machine schedule the connection itself, corresponding to a reliable, authenticated channel. We already stated above, that our modifications of $\mathsf{TH}_{\mathcal{H}}$ in

the "Send" transition carry over to the concrete implementation by using our simplified definition of reliable, non-authenticated channels between a and b. Being more precise, we introduced the simplified definition before the actual modification of $\mathsf{TH}_{\mathcal{H}}$, and the modification of $\mathsf{TH}_{\mathcal{H}}$ has been derived by that definition.

We will only sketch that the relation "at least as secure as" still holds for the modified system, because we would have to redo the whole original proof of [49], with only slight changes.

Proof (sketch). First of all, we set $L := \{a, b\}$ and $H := (\mathcal{I} \setminus \mathcal{H}) \cup \{\mathsf{A}\}$ representing the set of low and high level users, respectively. There are four possible communication "channels" on these sets that we have to take into account. A term of the form "H sends to L" means that an output of a high level user can result in on input of a low level user.

- H **sends to** H**.** In this case, both our ideal and real system have not been changed, so we obtain indistinguishable views in both systems.
- L **sends to** H**.** In this case, the only difference between the original and the modified systems is that initialization of a and b is done directly by $\mathsf{TH}_{\mathcal{H}}$, so the adversary is not able to initialize a connection between these two honest users by himself because initialization commands will be sorted out by construction of $\mathsf{TH}_{\mathcal{H}}$. In the real system, this implicit initialization of $\mathsf{TH}_{\mathcal{H}}$ exactly corresponds to a self-scheduled authenticated channel. In both systems, an initialization command will be output to the user, and immediately scheduled, along with the usual output to the adversary. Sending of messages has not been changed in both systems in this case, so we again obtain identical behaviours.
- H **sends to** L**.** The only modification in this case that initialization commands made by the adversary will always be sorted out. This has already been treated in the previous point, and it can moreover be easily achieved by a slight modification of the simulator of the original proof.
- L **sends to** L**.** The modifications in the initialization step have already been discussed in the second part. Obviously, the modified $\mathsf{TH}_{\mathcal{H}}$ and the self-scheduled authenticated channels yield identical views in both systems. In the "Send" transition, $\mathsf{TH}_{\mathcal{H}}$ immediately schedules the message to the corresponding user and sends a blinded copy to the adversary. In the real system reliable, non-authenticated channels do exactly the same: scheduling the message to the corresponding user and sending a blinded copy to the adversary. So we again have identical views of the user and the adversary in both systems.

■

7.4.3 Non-Interference Proof

In the following we will show that our system Sys^* fulfills its non-interference requirements given by the following flow policy. For two given elements $i_1, i_2 \in \mathcal{I} \cup \{\mathsf{A}\}$, we define $(S_{i_1}, S_{i_2}) \in \not\rightsquigarrow$ iff $i_1 \in (\mathcal{H} \setminus \{a, b\}) \cup \{\mathsf{A}\}$ and $i_2 \in \{a, b\}$. The flow policy is sketched in Figure 7.4.

Theorem 7.2 *(Non-Interference Properties of Sys^*)*

Let an arbitrary structure $(\{\mathsf{TH}_{\mathcal{H}}, \mathsf{M}_a^{\mathsf{n\_in}}, \mathsf{M}_b^{\mathsf{n\_in}}\}, S_{\mathcal{H}}) \in Sys^*$ be given. For the sake of readability, we set $\hat{M}_{\mathcal{H}} := \{\mathsf{TH}_{\mathcal{H}}, \mathsf{M}_a^{\mathsf{n\_in}}, \mathsf{M}_b^{\mathsf{n\_in}}\}$ in the following. Let a function ϕ_{Sys^*} be given that maps the structures $(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})$ of Sys^* to the flow policy $\mathcal{G}_{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})} = (\Delta_{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})}, \mathcal{E}_{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})})$ as defined above. The partition $\Delta_{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})}$ of $S_{\mathcal{H}}$ is defined by $\Delta_{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})} := \{S_i \mid i \in \mathcal{H}\} \cup \{\bar{S}\}$ with $S_i^c := \{\mathsf{out}_i?, \mathsf{in}_i!, \mathsf{in}_i^{\triangleleft}!\}$ for $i \in \mathcal{H}$ and $S_{\mathsf{A}} := \bar{S} = \mathsf{free}([\hat{M}_{\mathcal{H}}]) \setminus (\bigcup_{i \in \mathcal{H}} S_i)$. Then the system Sys^* fulfills ϕ_{Sys^*} perfectly. □

Before we turn our attention to the proof of Theorem 7.2, we present the following lemma.

Lemma 7.1 *By definition of the system, the following invariants hold for all possible runs of the configuration.*

1. *The collection* $\{\mathsf{M}_a^{\mathsf{n\_in}}, \mathsf{M}_b^{\mathsf{n\_in}}\}$*, i.e., the system* Sys_1*, is polynomial-time.*
2. *If* H_a *receives an input at* $\mathsf{out}_a?$*, it is of the form* $(\mathsf{rec\_init}, b)$ *or* $(\mathsf{receive}, b, m)$ *for an arbitrary* $m \in \Sigma^+$*. If* H_a *receives an input at* $\mathsf{master}_a?$*, it is sent by the master scheduler and is of the form* 1*.*
3. *No output of* $\mathsf{X}^{\mathsf{n\_in}}$ *at* $\mathsf{master}_a!$ *depends on inputs from other machines. Each machine is clocked equally often using a rotating clocking scheme. Furthermore, each output at a port* $\mathsf{p}^{\triangleleft}!$ *for* $\mathsf{p}^{\triangleleft}! \in S_a^c$ *and the scheduled message does only depend on prior outputs of* H_a *at port* $\mathsf{p}^{\mathsf{s}}!$ *and* $\mathsf{p}!$*.*
4. *If* H_a *receives a term of the form* $(\mathsf{rec\_init}, b)$ *at* $\mathsf{out}_a?$*, it is a direct consequence of the input* $(\mathsf{snd\_init})$ *sent by* H_b *(i.e., the scheduling sequence must have been* $\mathsf{H}_b, \mathsf{X}^{\mathsf{n\_in}}, \mathsf{M}_b^{\mathsf{n\_in}}, \mathsf{TH}_{\mathcal{H}}, \mathsf{M}_a^{\mathsf{n\_in}}, \mathsf{H}_a$ *or* $\mathsf{H}_b, \mathsf{X}^{\mathsf{n\_in}}, \mathsf{M}_b^{\mathsf{n\_in}}, \mathsf{M}_a^{\mathsf{n\_in}}, \mathsf{H}_a$*). This also implies that initializing a communication between* H_a *and* H_b *is not possible for the adversary, so there cannot be any replay attacks with initialization commands because they will be sorted out by* $\mathsf{TH}_{\mathcal{H}}$*.*
5. *If* H_a *receives a term of the form* $(\mathsf{receive}, b, m)$ *at* $\mathsf{out}_a?$*, it is a direct consequence (in the sense of Point 4) of the message* (send, m, a) *sent by* H_b*, so the scheduling sequence has been* $\mathsf{H}_b, \mathsf{X}^{\mathsf{n\_in}}, \mathsf{M}_b^{\mathsf{n\_in}}, \mathsf{TH}_{\mathcal{H}}, \mathsf{M}_a^{\mathsf{n\_in}}, \mathsf{H}_a$ *or* $\mathsf{H}_b, \mathsf{X}^{\mathsf{n\_in}}, \mathsf{M}_b^{\mathsf{n\_in}}, \mathsf{M}_a^{\mathsf{n\_in}}, \mathsf{H}_a$*. It is not possible for the adversary to pretend to be user* H_b*, and furthermore the number of received messages of this form and the number of messages sent by* H_b *to* H_a *are equal. Therefore, the adversary can neither replay these messages nor throw them away.*

The invariants also hold if we exchange the variables a *and* b*.* □

Proof.

1. The first part is quite obvious. The machine $\mathsf{M}_a^{\mathsf{n\_in}}$ ($\mathsf{M}_b^{\mathsf{n\_in}}$) has internal counters s_a and s'_a (s_b and s'_b) that are bounded by polynomials $s(k)$ and $s'(k)$ for a fixed security parameter k. At least one counter is always increased if $\mathsf{M}_a^{\mathsf{n\_in}}$ ($\mathsf{M}_b^{\mathsf{n\_in}}$) receives an input at port $\mathsf{in}_a?$ or $\mathsf{out}'_a?$ ($\mathsf{in}_b?$ or $\mathsf{out}'_b?$). An output at $\mathsf{p}_{\mathsf{M}_a}!$ ($\mathsf{p}_{\mathsf{M}_b}!$) must be a direct consequence of an input at $\mathsf{in}_a?$ or $\mathsf{out}'_a?$ ($\mathsf{in}_b?$ or $\mathsf{out}'_b?$) by construction of $\mathsf{M}_a^{\mathsf{n\_in}}$ ($\mathsf{M}_b^{\mathsf{n\_in}}$), so the number of messages sent over $\mathsf{p}_{\mathsf{M}_a}!$ ($\mathsf{p}_{\mathsf{M}_b}!$) can be at most $\max\{s(k), s'(k)\}$. If a counter reaches its bound the machine stops by construction, so the steps of the whole system Sys_1 is bounded by $\max\{s(k), s'(k)\}$. Thus, the collection $\{\mathsf{M}_a^{\mathsf{n\_in}}, \mathsf{M}_b^{\mathsf{n\_in}}\}$ is polynomial-time.

2. Part 2 follows by construction of $\mathsf{M}_a^{\mathsf{n\_in}}$ and $\mathsf{X}^{\mathsf{n\_in}}$. In the initialization transition the only possible output at $\mathsf{out}_a!$ is of the form $(\mathsf{rec\_init}, v)$. The check $v = b$ of $\mathsf{M}_a^{\mathsf{n\_in}}$ ensures $(\mathsf{rec\_init}, b)$. The only remaining step in which $\mathsf{M}_a^{\mathsf{n\_in}}$ may output something to H_a is the receive-message step. Outputs are of the form $(\mathsf{receive}, u, m)$. Again, $\mathsf{M}_a^{\mathsf{n\_in}}$ checks $u = b$ first so we can only have outputs of the form $(\mathsf{receive}, b, m)$. Thus, every output at port $\mathsf{out}_a!$ must have the desired form. The port $\mathsf{master}_a?$ is connected to the master scheduler, and it has to be of the form 1 by definition of the master scheduler $\mathsf{X}^{\mathsf{n\_in}}$.

3. The proof of part 3 is clear by construction of $\mathsf{X}^{\mathsf{n\_in}}$. At the start of the run, $\mathsf{X}^{\mathsf{n\_in}}$ schedules BIT_H and switches between case 2 and case 3 afterwards. In case 2, only the internal counter is checked and maybe $MaSc\_poly$ is increased, so no outputs from outside must be taken into account. Now assume that $\mathsf{X}^{\mathsf{n\_in}}$ outputs anything at $\mathsf{p}^{\triangleleft}!$ for $\mathsf{p}^{\triangleleft}! \in S_a^c$. This can only happen in case 3 if $\mathsf{X}^{\mathsf{n\_in}}$ receives exactly one input at one of the ports $\mathsf{p}^{\mathsf{s}}?$ for $\mathsf{p}^{\mathsf{s}}! \in \mathsf{ports}(\mathsf{H}_a)$. In this case it schedules the unique output port $\mathsf{p}^{\triangleleft}!$. Because of $\mathsf{p}! \in \mathsf{ports}(\mathsf{H}_a)$ only messages sent by H_a are scheduled which finishes this sub-part of the proof.

   Furthermore, note that neither the adversary nor an honest user can perform a clocked self-loop by definition. Moreover, the control will automatically come back to $\mathsf{X}^{\mathsf{n\_in}}$ after an arbitrary user is clocked by the system because users are forbidden to have any clockout ports by definition. If the adversary is scheduled it either has to do nothing or it has to schedule a machine of the system. In the first case the control immediately goes to the master scheduler, in the second one the machine of the system will either output nothing if one of its internal tests fails, or it finally schedules one of the honest users. In both cases $\mathsf{X}^{\mathsf{n\_in}}$ will be clocked again. This ensures that the master scheduler will always be scheduled after a constant number of steps, so it can in fact perform its rotation clocking scheme which clocks every machine equally often by definition.

4. This holds by construction of $\mathsf{M}_a^{\mathsf{n\_in}}$ and $\mathsf{TH}_{\mathcal{H}}$, and the previous part. First of all, note that H_a may only receive a term $(\mathsf{rec\_init}, b)$ if it has been output by $\mathsf{TH}_{\mathcal{H}}$ at $\mathsf{out}'_a!$ or by $\mathsf{M}_b^{\mathsf{n\_in}}$ at $\mathsf{p}_{\mathsf{M}_b}!$ in the previous step. The second case fulfills our requirements by construction of $\mathsf{M}_b^{\mathsf{n\_in}}$, because a message $(\mathsf{snd\_init})$ must have been output from H_b and scheduled by $\mathsf{X}^{\mathsf{n\_in}}$, so we can turn our attention to the first case. There are only two possibilities in which $\mathsf{TH}_{\mathcal{H}}$ may have output this term. The first case is initialization of user H_b, the second case is "Receive initialization". In the first case H_b outputs $(\mathsf{snd\_init})$, the master scheduler $\mathsf{X}^{\mathsf{n\_in}}$ schedules it (if H_b tells him what port to schedule, otherwise nothing is scheduled) and $\mathsf{TH}_{\mathcal{H}}$ directly outputs this term to $\mathsf{M}_a^{\mathsf{n\_in}}$ scheduling it immediately which fulfills our requirements. We will now show that $\mathsf{TH}_{\mathcal{H}}$ will not output anything in the other case. On input $(\mathsf{rec\_init}, b)$ at port $\mathsf{from\_adv}_a?$, $\mathsf{TH}_{\mathcal{H}}$ first checks $stopped_a^{\mathsf{spec}} = 0$, doing nothing at failure. After a successful test it checks $init_{b,a}^{\mathsf{spec}} = 0$. By our modification of $\mathsf{TH}_{\mathcal{H}}$ this can only hold if H_b does not have initialized itself, so $init_{b,b}^{\mathsf{spec}} = 0$ must hold. Because of $b \in \mathcal{H}$, $\mathsf{TH}_{\mathcal{H}}$ will not outputs anything which finishes the proof of this part.

5. The proof of this part can be done similar to the previous one. First of all, note that H_a may only receive a term $(\mathsf{receive}, b, m)$ if it has been output by $\mathsf{TH}_{\mathcal{H}}$ at $\mathsf{out}'_a!$ or by $\mathsf{M}_b^{\mathsf{n\_in}}$ at $\mathsf{p}_{\mathsf{M}_b}!$ in the previous step. The second case again fulfills our requirements by construction of $\mathsf{M}_b^{\mathsf{n\_in}}$. As in the previous step there

are are only two possibilities in which $\mathsf{TH}_{\mathcal{H}}$ may have output this term. The first case is sending messages of user H_b to user H_a, the second case is "Receive from honest party b". In the first case, H_b sends (send, m, a) which is again scheduled by $\mathsf{X}^{\mathsf{n\_in}}$. $\mathsf{TH}_{\mathcal{H}}$ directly outputs this term to $\mathsf{M}_a^{\mathsf{n\_in}}$ scheduling it immediately which fulfills our requirements. Furthermore, it increases the internal counter $msg\_out_{b,a}^{\mathsf{spec}}$. We will now finally show that $\mathsf{TH}_{\mathcal{H}}$ will not output anything in the second case. On input $(\mathsf{receive\_blindly}, b, i)$ at port $\mathsf{from\_adv}_a?$, $\mathsf{TH}_{\mathcal{H}}$ first does its usual initialization checks. We assume them to be successful, otherwise it outputs nothing anyway. It then checks $msg\_out_{b,a}^{\mathsf{spec}} = j$, if $(m, j) := deliver_{b,a}^{\mathsf{spec}}[i] \neq \downarrow$. However, the message counter $msg\_out_{b,a}^{\mathsf{spec}}$ is set to $msg\_in_{b,a}^{\mathsf{spec}} + 1$ after every sent message from b to a and $j \leq msg\_in_{b,a}^{\mathsf{spec}}$ always holds by construction of $\mathsf{TH}_{\mathcal{H}}$ for every $(m, j) := deliver_{b,a}^{\mathsf{spec}}[i] \neq \downarrow$. Hence, we have $j < msg\_out_{b,a}^{\mathsf{spec}}$ so $\mathsf{TH}_{\mathcal{H}}$ will not output anything which yields the desired result and finishes the proof of this part.

■

Proof. (Theorem 7.2) We have to show that Sys^* fulfills the non-interference requirement ϕ_{Sys^*}. Let an arbitrary structure $(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}) \in \mathsf{Conf}(Sys^*)$ be given so we have a flow policy $\mathcal{G}_{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})} = (\Delta_{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})}, \mathcal{E}_{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})})$ for this structure. Let now two arbitrary blocks $S_{i_1}, S_{i_2} \in \Delta_{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})}$ with $i_1 \in (\mathcal{H} \setminus \{a, b\}) \cup \{\mathsf{A}\}, i_2 \in \{a, b\}$ be given, so $(S_{i_1}, S_{i_2}) \in \not\rightsquigarrow$ must hold.

Let a non-interference configuration $conf_{i_1,i_2}^{n\_in} = (\hat{M}_{\mathcal{H}}, S_{\mathcal{H}}, U^{n\_in}, \mathsf{A})^{n\_in}$ for this structure be given. Without loss of generality we can assume $i_2 = a$ because of the symmetry of the flow policy.

Depending on the choice of the bit b we denote the two families of views of H_a by $view_{conf_{i_1,a}^{n\_in},0}(\mathsf{H}_a)$ and $view_{conf_{i_1,a}^{n\_in},1}(\mathsf{H}_a)$. Assume for contradiction that the probability of a correct guess $b = b^*$ is greater than $\frac{1}{2}$, which implies $view_{conf_{i_1,a}^{n\_in},0}(\{\mathsf{H}_a, \mathsf{H}_b\}) \neq view_{conf_{i_1,a}^{n\_in},1}(\{\mathsf{H}_a, \mathsf{H}_b\})$. First of all, we can exclude denial of service attacks applying Part 3 of the above lemma, so there has to be a first input at $\{\mathsf{H}_a, \mathsf{H}_b\}$ with different probability in both cases because Part 3 ensures that scheduling of messages sent by a user only depends on its own prior behaviour. We will now use the previous lemma to show that this cannot happen.

By Part 2 of Lemma 7.1 this input can only be of the form $(\mathsf{rec\_init}, u)$, $(\mathsf{receive}, u, m)$ at $\mathsf{out}_u?$, or 1 at $\mathsf{master}_u?$ for $u \in \{a, b\}$. We will in the following write $\bar{u}$ for the other emphasized user (i.e., $\bar{u} \in \{a, b\} \setminus \{u\}$). Assume this input to be of the first form. Now Part 4 implies that this input is a direct consequence of an input $(\mathsf{snd\_init})$ sent by the other emphasized user $\mathsf{H}_{\bar{u}}$. Hence, there had to be an input of $\mathsf{H}_{\bar{u}}$ with different probability in both cases which contradicts our assumption of the *first* different input, so there cannot be any influence from outside Sys_1. We therefore obtain identical probability distributions for possible inputs of H_a in both cases which yields the desired contradiction.

Now assume this input to be of the form $(\mathsf{receive}, u, m)$. By Part 5 the corresponding input $(\mathsf{send}, m, \bar{u})$ must have been sent directly by H_u with exactly the same message m. Furthermore, the underlying system for secure ordered channels ensures that the message has been sent exactly that often as H_a receives this input, so there cannot be any influence from outside because of the same reason as in the first case.

Finally, assume this input to be at port $\mathsf{master}_u?$. This input does not depend on arbitrary behaviours of other machines by Part 3 so we obtain identical probability

distributions again. Therefore, the views must in fact be identical in both cases so the probability of a correct guess $b = b^*$ is exactly $\frac{1}{2}$. Part 3 additionally ensures that every machine will be able to send messages if it wants to. This is not really necessary in our proof but it guarantees some kind of fairness a useful system ought to have. Thus, we have $Sys^* \models_{\mathsf{perf}} \phi_{Sys^*}$. ■

After proving the non-interference property for the ideal specification, we now concentrate on the concrete implementation.

Theorem 7.3 *(Non-Interference Properties of $Sys^\#$)* The real system $Sys^\#$ fulfills the non-interference property $\phi_{Sys^\#}$ computationally, with $\varphi_{Sys^\#}$ given as in theorem 7.1. In formulas, $Sys^\# \models^{\mathsf{poly}} \phi_{Sys^\#}$. □

Proof. Putting it all together, we know that the real implementation of secure message transmission with ordered channels is at least as secure as the specification with respect to computational indistinguishability of views. Moreover, we already sketched in Section 7.4.2 that this also holds for the modified systems. Using Part 1 of Lemma 7.1 we know that the system Sys_1 is polynomial-time, which is an essential precondition for applying the composition theorem in the computational case, so we have $Sys^\# \geq^{\mathsf{poly}} Sys^*$. Obviously, fulfillment of non-interference requirements in the perfect case implies fulfillment in the computational case, so we can conclude that the derived concrete implementation also fulfills its non-interference requirements with computational indistinguishability of views using theorem 7.1. ■

7.5 Conclusion

We have presented the first general definition of computational probabilistic non-interference in reactive systems (Section 7.2). Our approach is mainly motivated by the concept of simulatability which is fundamental for modern cryptography, and it might help to build a bridge between prior research in the field of information flow and systems involving real cryptographic primitives. We have shown that our definition behaves well under simulatability (Section 7.3), which enables modular proofs and step-wise refinement without destroying the non-interference properties. As an example fitting our definition we have presented an abstract specification of a cryptographic firewall guarding two honest users from their environment (Section 7.4). We have shown that the specification in fact fulfills the desired non-interference property. Moreover, we have presented a concrete implementation which we also showed to fulfill this property using our preservation theorem.

Chapter 8

Conclusion and Outlook

We successfully finished our attempt to build a bridge between the trustworthy proofs generated or verified by formal methods and the rigorous definitions and proofs of cryptography.

More precisely, we presented a general methodology how cryptographic protocols can be verified by formal proof tools, such that these proofs nevertheless maintain their sound cryptographic semantics, i.e., a leak in the proof could be used to break the underlying cryptography. Prior to this work, no such proof could be performed.

We believe that our approach paves the way for future protocol analysis, since it links real cryptography, i.e., cryptographic protocol involving real cryptographic primitives, and tool-assisted verification for the first time. Moreover, commonly accepted verification techniques might be used in our approach as well simply by using our sound abstractions instead of the usually assumed perfect cryptography.

The first step of our methodology considers the derivation of secure implementations of a given abstract specification. We showed that, besides the still necessary handmade proofs for suitably abstracting the cryptographic primitives, formally verified bisimulations in conjunction with the composition theorem of [49] are well-suited for this difficult and error-prone task. As an example, we presented an abstract specification of secure message transmission with ordered channels together with a secure implementation. The bisimulation which occurred in the security proof of the implementation has been formally verified using the theorem prover PVS.

The second step consists in the verification of the actual goals a protocol should fulfill. We expressed integrity properties in our underlying model and we showed that they are maintained under simulatability, i.e., properties proved for abstract specifications automatically carry over to their concrete implementations. This enables step-wise refinement and modular proofs, and it is an essential precondition for formal verification. Moreover, we showed that logic derivations among integrity properties are valid for the concrete implementation in a cryptographic sense, which makes them accessible for theorem provers. As an example, we formally validated our specification of secure message transmission with ordered channels, using the theorem proving system PVS, i.e., we showed that message reordering is in fact prevented. Now the verified property automatically carries over to the concrete implementation, which successfully finishes our attempt to verify both the security of the implementation and its actual goals.

As additional properties, we considered the important concepts of fairness, liveness, and information flow, which we expressed using the well-established method of probabilistic non-interference.

We showed that the standard definitions of integrity, fairness, liveness, and non-interference are not suited to cope with real cryptography, so we introduced new, more general definitions which circumvent this problem. These new, so-called polynomial definitions make these concepts accessible for real cryptographic protocols. More precisely, we restricted the definitions to polynomial length and included error probabilities. Similar to integrity properties, we showed that liveness and non-interference properties are preserved under simulatability, which again enables step-wise refinement and modular proofs. In case of liveness, we presented a specification and a secure implementation of secure message transmission with reliable channels, and we showed that both systems in fact fulfill the desired liveness property, i.e., reliability of messages. In case of non-interference, we presented a specification and a secure implementation of a cryptographic firewall guarding two honest user from their environment.

If we consider our approach as a whole, i.e., combining the results of [49] and this work, we can state the following results: we successfully finished presenting the first approach for cryptographic protocol verification which is sound with respect to the underlying cryptographic primitives, and that allows abstractions suitable for formal proof systems. These abstractions are easy to use and can be combined to large protocols very easily. Moreover, these abstraction can be refined step-wise in order to obtain formally verified concrete systems, and formal proofs made for the abstract specification automatically carry over to the concrete implementation.

Concerning future work, there are innumerous things to do. Obviously, the security of our implementation which we verified using PVS is still based on paper-and-pencil proofs such as the composition theorem, the transitivity lemma, or the security proof of the primitive used. Hence, one future step could be the verification of those theorems using formal proof systems. However, we are aware of the difficulty of this task because we have to handle probabilism and also computational variants. After verifying the basic theorems of our model, we are likely to turn our attention to more sophisticated cryptographic primitives like commitments. By now, there is already progress at our group at Saarland University and IBM Zurich on designing a cryptographic library which should provide sound abstractions of common cryptographic primitives including common concepts for standard protocol design like nonces, timestamp, and so on. On the long run, we hope to be able to verify really large systems which are commonly used in practice (e.g., e-commerce architectures), and which are essentially based on these concepts.

Concerning liveness properties of cryptographic protocols, we might extend our definition of polynomial liveness to multiple run-empty phases, i.e., we might consider multiple good events and continuous interaction of users and the adversary.

Finally, we may include additional properties in our definition of non-interference. So far, we restricted ourselves to transitive flow policies. However, there are several interesting examples of intransitive flow policies, e.g., the original work by Haigh and Young [25] who used it to formalize security for type enforcement, or the more recent work by Roscoe [51] who cast intransitive non-interference within CSP. If we consider cryptographic protocols it might be of great interest to state that there can only be information flow between two honest users iff special additional users must have been involved. As a possible application of such a definition we can consider secret sharing. Assume that the first user shares a secret among some other users. Then a second user might not be able to send information to the first user all by itself, but if it gets some help of the remaining users, the secret can be restored resulting in the desired information flow. Moreover, the concept of downgrading certain information which we

already sketched briefly in the corresponding chapter seems to be another important goal to strive for. A possible way to express this would be to include conditional probabilities in our usual definition of non-interference but future research is needed in this area.

Appendix A

Postponed Proofs

A.1 From Section 3.1

Proof. (Lemma 3.1). Let a configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A}) \in \mathsf{Conf}(Sys)$ be given.
Construction of $\mathsf{H}_{\mathsf{A\_H}}$:
According to Figure 3.1, we define a new honest user $\mathsf{H}_{\mathsf{A\_H}}$ using H as a blackbox submachine as follows. Its ports are given by

- $\{\mathsf{p} \mid \mathsf{p} \in \mathsf{ports}(\mathsf{H}) \wedge \mathsf{p}^c \in S\}$, i.e., it has the same ports for connecting to the specified ports of the structure as the original user H.
- $\{\mathsf{p}!, \mathsf{p}? \mid (\mathsf{p}! \in \mathsf{ports}(\mathsf{H}) \wedge \mathsf{p}!^c \notin S) \vee (\mathsf{p}? \in \mathsf{ports}(\mathsf{H}) \wedge \mathsf{p}?^c \notin S)\}$, i.e., it has the same set of ports for connecting to the adversary, but additionally it has the corresponding input, or output ports formerly owned by the adversary. This yields self-loops, see Figure 3.1, which will be essential for achieving identical views. Our idea is that the contents of these so-called "self-loop-buffers" will always equal the original buffers between H and A.
- $\{\mathsf{p}^{\triangleleft}! \mid \mathsf{p}^{\triangleleft}! \in \mathsf{ports}(\mathsf{H})\}$, i.e., is has the same set of clockout ports.
- $\{\mathsf{p}_{\mathsf{A\_H}}?, \mathsf{p}_{\mathsf{H\_A}}!, \mathsf{p}_{\mathsf{H\_A}}{}^{\triangleleft}!\}$, i.e., the special duplex port for communication with the adversary.

Its behaviour is defined as follows.

- If it gets a non-empty input m at a port $\mathsf{p}?$ from the system (i.e., $\mathsf{p}?^c \in S$) it applies $\delta_{\mathsf{H}}(s, \mathcal{I}_{\mathsf{p}?=m})$ where s denotes the current state of the blackbox submachine, yielding a tuple $(s', \mathcal{O})$. $\mathsf{H}_{\mathsf{A\_H}}$ now outputs this tuple at its own output ports and switches to the same state s'.[1]
- If it gets a non-empty input m at a self-loop port $\mathsf{p}?$ with $\mathsf{p}? \in \mathsf{ports}(\mathsf{H})$ (which means that this has been a connection for *receiving* messages either from the adversary or from itself), it applies the state transition function δ_{H} on $(s, \mathcal{I}_{\mathsf{p}?=m})$ yielding a tuple $(s', \mathcal{O})$. Now, it again outputs $\mathcal{O}$ and switches to state s'.

[1] Note, that this is indeed possible, because every output port of H is also an output port of $\mathsf{H}_{\mathsf{A\_H}}$ by construction.

- If it gets a non-empty input m at a self-loop port $\mathsf{p}?$ with $\mathsf{p}? \notin \mathsf{ports}(\mathsf{H})$ (which means that this has been a connection for *sending* messages to the adversary), it outputs $(\mathsf{p}?, m)$ at $\mathsf{p}_{\mathsf{H\_A}}!$, 1 at $\mathsf{p}_{\mathsf{H\_A}}{}^{\triangleleft}!$. Formally, we consider an efficiently computable mapping $\phi : \Sigma^+ \times \Sigma^+ \mapsto \Sigma^+$ so that decomposition is unique and efficiently computable. Moreover, we assume that there is an element $\top \in \Sigma^+$ with $\top \notin \mathsf{Im}(\phi)$. Intuitively, $\mathsf{H}_{\mathsf{A\_H}}$ tells the adversary $\mathsf{A}_{\mathsf{A\_H}}$ that a message m would arrive at $\mathsf{p}?$ in the original configuration $conf$.

- If $\mathsf{H}_{\mathsf{A\_H}}$ gets a non-empty input m' at $\mathsf{p}_{\mathsf{A\_H}}?$, it tries to decompose m' into the form $(\mathsf{p}?, m)$ with $\mathsf{p}? \in \mathsf{ports}(\mathsf{H})$ and $m \in \Sigma^+$, doing nothing at failure. In case of success, it outputs m at $\mathsf{p}!$, $\top$ at $\mathsf{p}_{\mathsf{H\_A}}!$, and 1 at $\mathsf{p}_{\mathsf{H\_A}}{}^{\triangleleft}!$. This case ensures that messages sent by the adversary are delivered to the corresponding self-loop-buffers. After that, $\mathsf{H}_{\mathsf{A\_H}}$ explicitly gives the control back to the adversary by scheduling the special message $\top$. Thus, the still to define adversary $\mathsf{A}_{\mathsf{A\_H}}$ can send messages intended for the honest user one by one scheduling them immediately. $\mathsf{H}_{\mathsf{A\_H}}$ writes the message into the corresponding buffer and gives the control back to the adversary.

Construction of $\mathsf{A}_{\mathsf{A\_H}}$:
In order to obtain the adversary $\mathsf{A}_{\mathsf{A\_H}}$, the original adversary A is modified in a similar way. If $\mathsf{A}_{\mathsf{A\_H}}$ receives an arbitrary input from the system it calls its internal blackbox function δ_{A} on this input. If it receives an input $m' \neq \top$ at $\mathsf{p}_{\mathsf{H\_A}}?$ it tries to decompose it into the form $(\mathsf{p}?, m)$ with $\mathsf{p}? \in \mathsf{ports}(\mathsf{A})$. If this fails it does nothing, otherwise it calls $\delta_{\mathsf{A}}(s, \mathcal{I}_{\mathsf{p}?=m})$ yielding a tuple $(s', \mathcal{O})$. The case $m' = \top$ is defined below.

Now, $\mathsf{A}_{\mathsf{A\_H}}$ checks whether there are non-empty outputs at a port $\mathsf{p}!$ with $\mathsf{p}? \in \mathsf{ports}(\mathsf{H})$. If this does not hold it simply outputs the tuple $\mathcal{O}$. This corresponds to the case that all buffers between H and A remain unchanged in $conf$, so we do not have to worry about this case.

Otherwise, it stores all such nonempty values at ports $\mathsf{p}!$ with $\mathsf{p}? \in \mathsf{ports}(\mathsf{H})$ in internal arrays, so that every array corresponds to exactly one port. Let now $\mathsf{p}?$ be the first port which has a non-empty array, then the first entry m of the array is removed and $\mathsf{A}_{\mathsf{A\_H}}$ outputs $(\mathsf{p}?, m)$.

If $\mathsf{A}_{\mathsf{A\_H}}$ receives an input $\top$ at $\mathsf{p}_{\mathsf{H\_A}}?$, it again looks for the first port with a non-empty array yielding a new output of the form $(\mathsf{p}'?, m')$ again and so on until all arrays are empty. If all arrays have finally been emptied the rest of the original tuple $\mathcal{O}$ is output.

Putting it all together, messages sent to H are delivered one by one to $\mathsf{H}_{\mathsf{A\_H}}$ which outputs them to their intended buffers. After every message, $\mathsf{H}_{\mathsf{A\_H}}$ schedules $\mathsf{A}_{\mathsf{A\_H}}$ again which can now send the next message and so on, until all messages are finally delivered. This gives us a new configuration

$$conf_{\mathsf{A\_H}} = (\hat{M}, S, \mathsf{H}_{\mathsf{A\_H}}, \mathsf{A}_{\mathsf{A\_H}}) \in \mathsf{Conf}(Sys).$$

Obviously, $conf_{\mathsf{A\_H}} \in \mathsf{Conf}_{\mathsf{A\_H}}(Sys)$ holds which finishes the first part of the proof.
Proof of indistinguishability:
We now have to show that the view of H is indistinguishable in both configurations. First of all, note that the newly defined machines $\mathsf{H}_{\mathsf{A\_H}}$ and $\mathsf{A}_{\mathsf{A\_H}}$ simply forward inputs made by the system to their blackbox submachine. We will now prove by induction over the run that the contents of the buffers between H and A in $conf$ and the contents of the corresponding "self-loop" buffers of $\mathsf{H}_{\mathsf{A\_H}}$ in $conf_{\mathsf{A\_H}}$ are always equal at every

time either H, A, or a machine of the system switches. We will use this to show that the views of H are identical in both configurations.

At the start of the run, all these buffers are empty in both configurations yielding a correct start of our induction.

We first take a look at buffers for sending messages from H to A. Assume, that H outputs an arbitrary tuple $\mathcal{O}$ in $conf$. In $conf_{\mathsf{A\_H}}$, $\mathsf{H}_{\mathsf{A\_H}}$ outputs the same tuple at the same ports which surely yields the desired result.

If such a buffer $\widetilde{\mathsf{p}}$ is scheduled with input i (it does not matter which machine has the corresponding clockout port $\mathsf{p}^{\triangleleft}!$), the i-th element m is sent to the recipient. In the second configuration, the i-element m' is delivered to its original sender $\mathsf{H}_{\mathsf{A\_H}}$. Applying our induction hypothesis, $m = m'$ must hold. Now, $\mathsf{H}_{\mathsf{A\_H}}$ outputs a tuple $(\mathsf{p}?, m)$ at $\mathsf{p}_{\mathsf{A\_H}}!$, scheduling it immediately, $\mathsf{A}_{\mathsf{A\_H}}$ decomposes this tuple into port name and contents and applies its blackbox function using the correct input m at the correct port $\mathsf{p}?$. The buffers remain mapped to each other between the two clockings of H and A, and this step obviously yields identical views for H and A_1 because they neither notice that the message has been sent back to its sender $\mathsf{H}_{\mathsf{A\_H}}$ nor that the message has been changed to a tuple and has been decomposed again later.

We finally have to consider sending messages from A to H. Assume $\widetilde{\mathsf{p}}$ to be a buffer for sending messages from A to H. This case is slightly more complicated. If A makes an arbitrary output $\mathcal{O}$ in $conf$, the messages are immediately stored in the desired buffers. In $conf_{\mathsf{A\_H}}$, $\mathsf{A}_{\mathsf{A\_H}}$ sends the first of these messages to $\mathsf{H}_{\mathsf{A\_H}}$ which outputs it at the corresponding port, so we now have equal contents of this buffer in both configurations by our induction hypothesis. Now, it gives back control to $\mathsf{A}_{\mathsf{A\_H}}$ and the game continues with the second message and so on until all outputs to H are considered and the contents of every such buffers are equal again. Finally, it outputs the remaining components, i.e., outputs to the system and to itself, and the first scheduling output. Scheduling of such a buffer $\widetilde{\mathsf{p}}$ now obviously yields identical results in both configurations by our induction hypothesis because the affected buffers between H and A have already been filled.

Moreover, outputs of H and A to the system are simply forwarded by $\mathsf{H}_{\mathsf{A\_H}}$ and $\mathsf{A}_{\mathsf{A\_H}}$, respectively, which yields identical outputs for the machines of the system. Thus, we obtain identical views of H and A, and furthermore identical behaviours for the machines of the considered structure. Altogether, this yields

$$view_{conf}(\mathsf{H}) = view_{conf_{\mathsf{A\_H}}}(\mathsf{H}).$$

We finally have to show that both $\mathsf{H}_{\mathsf{A\_H}}$ and $\mathsf{A}_{\mathsf{A\_H}}$ are polynomial-time if H and A are polynomial-time. Both machines $\mathsf{H}_{\mathsf{A\_H}}$ and $\mathsf{A}_{\mathsf{A\_H}}$ either only forward inputs and outputs to their blackbox submachines or they deliver the messages to the corresponding buffers iteratively as described above. This needs at most $|\mathsf{ports}(\mathsf{H})|$ steps (i.e., a constant number of steps), so both machines perform only a constant number of steps between two successive clockings of their submachines. Thus, both machines must be polynomial-time if the original user and adversary has been polynomial-time which finishes the proof. ∎

Proof. (Lemma 3.2) We first reverse our construction on $\mathsf{H}_{\mathsf{A\_H}}$ to obtain the original user H again. Now, we define the new adversary A_2 as follows. It has the same set of ports as A_1 but instead of the ports $\mathsf{p}_{\mathsf{H\_A}}?$, $\mathsf{p}_{\mathsf{A\_H}}!$ and $\mathsf{p}_{\mathsf{A\_H}}{}^{\triangleleft}!$, it has the "original" ports for connecting to the honest user, i.e.,

$$\{\mathsf{p} \mid \mathsf{p}^C \in \mathsf{ports}(\mathsf{H}) \wedge \mathsf{p} \notin \mathsf{ports}(\hat{M} \cup \{\mathsf{H}\})\}.$$

Furthermore, it has three special ports $\{\mathsf{p}_{\mathsf{self}}?, \mathsf{p}_{\mathsf{self}}!, \mathsf{p}_{\mathsf{self}}{}^{\triangleleft}!\}$ for a clocked self-loop.

Its behaviour can be defined very simple: if it gets a non-empty input from the system it uses A_1 as a blackbox on this input yielding a new state and an output tuple $\mathcal{O}$. If it gets a non-empty input m at a port $\mathsf{p}?$ connected to the honest user, it applies δ_{A_1} with $\mathcal{I}_{\mathsf{p}_{\mathsf{H\_A}}?=(\mathsf{p}?,m)}$.

So far, inputs of A_1 are obviously the same in both configurations. Now, let an arbitrary output tuple $\mathcal{O}$ be given. If $\mathsf{p}_{\mathsf{A\_H}}! = \epsilon$ and $\mathsf{p}_{\mathsf{A\_H}}{}^{\triangleleft}! = \epsilon$ hold it simply outputs $\mathcal{O}$. In this case there are only outputs to the system and to itself which obviously yields identical views for A_1, the machines of $\hat{M}$ and especially H. If $\mathsf{p}_{\mathsf{A\_H}}! = m \neq \epsilon$ it appends m at an internal list over Σ^+. This internal list will be used to model the (now missing) buffer $\widetilde{\mathsf{p}_{\mathsf{A\_H}}}$ of $conf_{\mathsf{A\_H},*}$, i.e., their contents should always be equal. If $\mathsf{p}_{\mathsf{A\_H}}{}^{\triangleleft}! = i \neq \epsilon$ it first removes the i-th element from this array doing nothing in case of fewer elements. In $conf_{\mathsf{A\_H},*}$, the scheduled buffer would also have failed so the control goes to the master scheduler in both configurations. A_2 now tries to decompose this element into the form $(\mathsf{p}?, m)$ with $\mathsf{p}? \in \mathsf{ports}(\mathsf{H})$, doing nothing at failure. In $conf_{\mathsf{A\_H},*}$ the honest user $\mathsf{H}_{\mathsf{A\_H}}$ would also have failed to decompose the message in this case so the control again goes to the master scheduler in both configurations. In case of success, it outputs m at $\mathsf{p}!$ and schedules itself (using the clocked self-loop $\mathsf{p}_{\mathsf{self}}$, e.g., it outputs 1 at $\mathsf{p}_{\mathsf{self}}!$ and 1 at $\mathsf{p}_{\mathsf{self}}{}^{\triangleleft}!$). Now, it applies its blackbox function δ_{A_1} with $\mathcal{I}_{\mathsf{p}_{\mathsf{H\_A}}?=\top}$ which corresponds to the behaviour of $\mathsf{H}_{\mathsf{A\_H}}$ in $conf_{\mathsf{A\_H},*}$ (the control is explicitly given back to the adversary). Intuitively, A_2 now "plays the role" of $\mathsf{H}_{\mathsf{A\_H}}$. Obviously, this results in identical views for A_1. Moreover, we have identical behaviours with respect to the environment, i.e., the machines of the structure.

It is now easy to see that we also obtain identical views for H in both configurations. More formally, we could as usual consider inputs and outputs of H and A_1 in both configurations and show that the contents of the buffers between H and A_2 in $conf_*$ and the contents of the self-loop buffers of $\mathsf{H}_{\mathsf{A\_H}}$ in $conf_{\mathsf{A\_H},*}$ are always equal which finally results in identical views of the honest user in both configurations. This can be proven similar as in the precedent proof so we will omit this straightforward but tedious exercise this time. We finally obtain

$$view_{conf_{\mathsf{A\_H},*}}(\mathsf{H}) = view_{conf_*}(\mathsf{H}).$$

As in the previous proof, A_2 mainly forwards inputs and outputs to and from the blackbox submachine. Now and then, it has to perform a clocked self-loop, which only takes a constant number of steps. Moreover, it uses its blackbox submachine at least once between two successive clocked self-loops. Hence, A_2 is polynomial-time if the original A_1 is polynomial-time. Moreover, H is polynomial-time if $\mathsf{H}_{\mathsf{A\_H}}$ is polynomial-time (cf. the previous proof). Thus, $conf_*$ is a polynomial-time configuration if $conf_{\mathsf{A\_H},*}$ is polynomial-time, which finishes the proof. ∎

A.2 From Section 3.2

Proof. (Lemma 3.3) Let S_{A} denote the set of specified ports the adversary connects to, i.e.,

$$S_{\mathsf{A}} := \{p \mid p \in S \setminus \mathsf{ports}(\mathsf{H})^c\}.$$

Roughly speaking, we will define a new machine H_1 which is inserted between the system and the adversary such that H_1 uses all ports of S_{A}. Combination of H_1 and H

will yield the new honest user $\mathsf{H_s}$. However, we will at first concentrate on the machine $\mathsf{A_s}$.

If the configuration $conf$ is polynomial-time, let the adversary A be bounded by $L(k)$ for a polynomial L and the security parameter k. We now define the new adversary $\mathsf{A_s}$ of $conf_\mathsf{s}$ starting with its ports.

- First of all, every port $p \in \mathsf{ports(A)}$ that does not connect to a specified port, i.e. $p^c \notin S_\mathsf{A}$, is also a port of $\mathsf{A_s}$.
- For every simple port $p \in \mathsf{ports(A)}$ with $p^c \in S_\mathsf{A}$, $\mathsf{A_s}$ has a port p' of the same kind.
- For every clockout port $\mathsf{p}^{\triangleleft}! \in \mathsf{ports(A)}$ that connects to the specified ports, i.e. $\mathsf{p}^{\triangleleft}!^c \in S_\mathsf{A}$, $\mathsf{A_s}$ has a clockout port $\mathsf{p_{cr}}^{\triangleleft}!$ and an additional output port $\mathsf{p_{cr}}!$.[2]
- $\mathsf{A_s}$ has additional ports $\mathsf{p_{A_s}}?, \mathsf{p_{H_1}}!, \mathsf{p_{H_1}}^{\triangleleft}!$ which will be needed for synchronizing the communication with $\mathsf{H_1}$ and ports $\mathsf{p_{mask\_back}}?, \mathsf{p_{mask}}!, \mathsf{p_{mask}}^{\triangleleft}!$ needed to make the machine $\mathsf{H_1}$ polynomial-time in case of a polynomial-time configuration $conf$.

We assume without loss of generality that all these primed and additional ports are new ports of the configuration. Internally, $\mathsf{A_s}$ maintains an array $\mathcal{O}'\_save = (O'\_save_{\mathsf{p}^{\triangleleft}!})_{\mathsf{p}^{\triangleleft}! \in \mathsf{ports(A_s)}}$ over Σ^+ initialized with ϵ everywhere and two arrays $(out\_buff_{\mathsf{p}!})_{\mathsf{p}! \in S_\mathsf{A}^c}$ and $(masked_{\mathsf{p}?})_{\mathsf{p}? \in S_\mathsf{A}^c}$ over $\{0,1\}$ initialized with 0 everywhere.

The array out_buff will be used to indicate the buffers between A and the corresponding specified ports of the system which have nonempty contents. The array $masked$ will be used to explicitly tell $\mathsf{H_1}$ which input ports it has to "cut off", i.e., which ports it has to mask with a length bound 0.

The behaviour of $\mathsf{A_s}$ is now defined as follows. On an arbitrary given output $\mathcal{O} = (O_p)_{p \in \mathsf{ports(A)}}$ and the given state s' of the blackbox A, the corresponding output $\mathcal{O}' = (O'_p)_{p \in \mathsf{ports(A_s)}}$ of $\mathsf{A_s}$ is derived by the following algorithm. Initially, all components of $\mathcal{O}'$ are set to ϵ. They are automatically reset to ϵ at every call of δ_A.

- *Preliminary Step:* First of all, $\mathsf{A_s}$ checks whether A masked one of its own input ports connected to the specified ports using a zero length bound, which it can easily do, because it knows the current state s' of the blackbox. For every masked input port $\mathsf{p}? \notin S_\mathsf{A}^c$, i.e., a port connected to an unspecified port of the system, it masks this input too; for every non-masked input port of this kind, it sets the length bound to the runtime of A in the polynomial case and to infinity in the remaining cases. For every masked input port $\mathsf{p}? \in S_\mathsf{A}^c$, it sets $masked_{\mathsf{p}?} := 1$. Afterwards, it sets $\mathcal{O}\_save := \mathcal{O}$ (i.e., it saves the whole output tuple, since it will need the tuple after it has been scheduled by $\mathsf{H_1}$ again) and encodes the whole array $masked$ into $c \in \Sigma^*$. Finally, it outputs c at $\mathsf{p_{mask}}!$, 1 at $\mathsf{p_{mask}}^{\triangleleft}!$. Informally speaking, $\mathsf{A_s}$ has to tell $\mathsf{H_1}$ which ports it should mask, so it stores them in the array $masked$ and sends the whole array to $\mathsf{H_1}$.
- *Step 1:* $\mathsf{p}! \notin S_\mathsf{A}^c$: At first, $\mathsf{A_s}$ simply goes through the tuple and sets $O'_{\mathsf{p}!} = O_{\mathsf{p}!}$ for every port $\mathsf{p}!$ with $\mathsf{p}! \notin S_\mathsf{A}^c$. This case ensures that outputs to itself, to the system, and to the original honest user H will simply be forwarded.

[2] The index cr serves as an abbreviation for "clocking request". These ports will later be used to tell $\mathsf{H_1}$ which buffer it has to schedule.

- *Step 2:* $\mathsf{p}! \in S^c_{\mathsf{A}}$: Then, $\mathsf{A_s}$ goes through the tuple and sets $O'_{\mathsf{p}'!} = O_{\mathsf{p}!}$ for every port $\mathsf{p}!$ with $\mathsf{p}! \in S^c_{\mathsf{A}}$. If $O_{\mathsf{p}!} \neq \epsilon$, $\mathsf{A_s}$ additionally sets $out\_buff_{\mathsf{p}'!} := 1$, i.e., it stores which buffers between $\mathsf{A_s}$ and H_1 have nonempty content.

So far we have considered outputs at the simple ports of A. Now $\mathsf{A_s}$ goes through the tuple and searches for the first nonempty output at a clockout port $\mathsf{p}^{\triangleleft}!$.

- *Step 3:* $\mathsf{p}^{\triangleleft}! \in S^c_{\mathsf{A}}$: If A outputs c at a clockout port $\mathsf{p}^{\triangleleft}! \in S^c_{\mathsf{A}}$, A' encodes c and the whole array out_buff_p into $c' \in \Sigma^+$. It then sets $O'_{\mathsf{p_{cr}}!} = c'$, $O'_{\mathsf{p_{cr}}^{\triangleleft}!} = 1$, and $out\_buff_p = 0$ for all elements of the array and outputs $\mathcal{O}'$. Informally speaking, $\mathsf{A_s}$ tells H_1 what buffer have nonempty contents at the moment, and that it should schedule the c-th message of buffer $\widetilde{\mathsf{p}}$ afterwards.

- *Step 4:* $\mathsf{p}^{\triangleleft}! \notin S^c_{\mathsf{A}}$ *or no non-empty clock output at all* : If A outputs c at $\mathsf{p}^{\triangleleft}!$ with $\mathsf{p}^{\triangleleft}!^c \notin S_{\mathsf{A}}$, $\mathsf{A_s}$ encodes the whole array out_buff_p into $c' \in \Sigma^+$ as in the previous step but containing the number 0 instead of the number c. It then sets $O'_{\mathsf{p_{H_1}}!} := c'$, $O'_{\mathsf{p_{H_1}}^{\triangleleft}!} := 1$, $O'\_save_{\mathsf{p}^{\triangleleft}!} := c$, and $out\_buff_p = 0$ for all elements of the array and outputs $\mathcal{O}'$.

  We again briefly sketch the intuition behind this case. Messages intended for the system are directly output, but no message is immediately scheduled. Again, $\mathsf{A_s}$ tells H_1 all necessary information for delivering messages to the specified ports, but additionally, it stores which buffer it has to schedule afterwards. Anticipating, H_1 will give back control to $\mathsf{A_s}$ by construction after he delivered the messages to the specified ports, so $\mathsf{A_s}$ will be able to schedule the desired buffer $\widetilde{\mathsf{p}}$.

  If there is no nonempty clock output, $\mathsf{A_s}$ acts identically but sets $O'\_save_{\mathsf{p}^{\triangleleft}!} := \epsilon$ instead. This ensures that no buffer will be scheduled after the control comes back from H_1 to $\mathsf{A_s}$, so the master scheduler will be scheduled just as in the original configuration $conf$.

The behaviour of $\mathsf{A_s}$ on external inputs can be described quite simply.

- If $\mathsf{A_s}$ receives an input 1 at $\mathsf{p_{A_s}}?$ (i.e., the machine H_1 gives back the control), it simply outputs $\mathcal{O}'\_save$ and sets $O'\_save_{\mathsf{p}^{\triangleleft}!} = \epsilon$ afterwards for all elements of the array. This case can only occur as a direct consequence of Step 4 of the above algorithm. Inputs at other ports are simply forwarded to their corresponding ports of A.

- If $\mathsf{A_s}$ receives an input 1 at $\mathsf{p_{mask\_back}}?$ it sets all components of $masked$ back to 0 and $\mathcal{O} := \mathcal{O}\_save$ and proceeds with Step 1.

- If A enters final state, we define that $\mathsf{A_s}$ finishes the delivering of messages and enters final state too. More precisely, it outputs its tuple derived by the above algorithm and stops. If Step 4 applies, it additionally waits for a nonempty input at $\mathsf{p_{A_s}}?$, outputs the tuple $\mathcal{O}'\_save$, i.e., the scheduling of the desired buffer, and enters final state after that.

Note, that $\mathsf{A_s}$ obviously can only do a polynomial number of steps between two successive calls of δ_{A} by construction which yields a polynomial-time adversary $\mathsf{A_s}$ again if A is polynomial.

We can now turn our attention to the machine H_1 which is defined as follows. Its ports are given by

- $\{\mathsf{p} \mid \mathsf{p}^c \in S_{\mathsf{A}}\}$: Ports for connecting to the specified ports S_{A}.
- $\{\mathsf{p}'?, \mathsf{p}'^{\triangleleft}! \mid \mathsf{p}?^c \in S_{\mathsf{A}}\}$: Input ports for connecting to A_s.
- $\{\mathsf{p}'!, \mathsf{p}'^{\triangleleft}! \mid \mathsf{p}!^c \in S_{\mathsf{A}}\}$: Output ports for connecting to A_s.
- $\{\mathsf{p}_{\mathsf{cr}}? \mid \mathsf{p}^{\triangleleft}!^c \in S_{\mathsf{A}}\}$: Input ports for clocking requests of A_s.
- $\{\mathsf{p}_{\mathsf{H}_1}?, \mathsf{p}_{\mathsf{A}_\mathsf{s}}!, \mathsf{p}_{\mathsf{A}_\mathsf{s}}{}^{\triangleleft}!\}$: Ports for synchronization with A_s.
- $\{\mathsf{p}_{\mathsf{mask}}?, \mathsf{p}_{\mathsf{mask\_back}}!, \mathsf{p}_{\mathsf{mask\_back}}{}^{\triangleleft}!\}$: Ports for making explicit changes of length bounds. As already described above, these ports will be used for masking certain inputs.

Internally, H_1 maintains an array $(\mathit{buff\_coll}_{\mathsf{p}?})_{\mathsf{p}? \in S_{\mathsf{A}}}$ over Σ^+ initialized with ϵ everywhere. The behaviour of H_1 is defined as follows.

- If H_1 receives an input c at $\mathsf{p}_{\mathsf{mask}}?$ it decomposes c into the array masked again. For every $\mathit{masked}_{\mathsf{p}?} = 1$ it masks the input port $\mathsf{p}?$ using a zero length bound. For every $\mathit{masked}_{\mathsf{p}?} = 0$ it sets the length bound of $\mathsf{p}?$ to the runtime of A in the polynomial case; otherwise, it sets it to infinity.
- If H_1 receives an input c at a port $\mathsf{p}?$, it outputs c at $\mathsf{p}'!$, 1 at $\mathsf{p}'^{\triangleleft}!$. This case ensures that outputs made by system are simply forwarded to the adversary.
- If H_1 receives an input c' at $\mathsf{p}_{\mathsf{cr}}?$, it decomposes c' into its original form $c' = c, (\mathit{out\_buff}_{\mathsf{p}!})_{\mathsf{p}! \in S_{\mathsf{A}}^c}$.
  - In case $c \neq 0$, it does the following: For every element $\mathit{out\_buff}_{\mathsf{p}!} \neq 0$ it schedules the message stored in $\widetilde{\mathsf{p}'}$ and saves them in $\mathit{buff\_coll}_{\mathsf{p}?}$.[3] After that, H_1 outputs the array $\mathit{buff\_coll}_{\mathsf{p}?}$ to the corresponding output ports $\mathsf{p}!$ and removes these elements from the array (which yields an empty array again). Additionally, it outputs c at $\mathsf{p}^{\triangleleft}!$ (the corresponding clocking port for requests at $\mathsf{p}_{\mathsf{cr}}?$).
  - In case $c = 0$, it collects all messages stored in the buffers $\widetilde{\mathsf{p}'}$ in $\mathit{buff\_coll}_{\mathsf{p}?}$ again as in the previous step. Finally, it outputs these messages at their corresponding ports and 1 at $\mathsf{p}_{\mathsf{A}_\mathsf{s}}!$, 1 at $\mathsf{p}_{\mathsf{A}_\mathsf{s}}{}^{\triangleleft}!$. This case ensures that the adversary A will be scheduled again, so he can eventually schedule its desired buffer (cf. Step 4 of the description of A_s).

If the configuration conf is polynomial-time, we let H_1 also stop after a polynomial number of steps. A possible polynomial bound can simply be derived if you consider that H_1 has to make less than $|\mathsf{ports}(\mathsf{A}_\mathsf{s})|$ outputs for collecting messages from the nonempty buffer. These messages are stored in the corresponding arrays and finally output as a tuple. The number of ports is finite and does not depend on the security parameter k, so the number of steps which H_1 performs between two successive clockings of itself in every run is constant, because masking of input ports is done not only by A but also H_1. Moreover, H_1 can only be clocked either by the system or by the adversary. If it is clocked by the system it immediately clocks A_s which has to be polynomial-time if A is polynomial-time as we showed above. Thus, H_1 can only perform a constant

[3] This is indeed possible, because the scheduled buffer will schedule H_1 again by construction if it has a nonempty output. This will always be the case, since H_1 will only schedule buffers which he knows to be nonempty.

number of steps between two successive clockings of $\mathsf{A_s}$. If we denote this constant by cst, H_1 simply stops after $\mathsf{cst} \cdot L_{\mathsf{A_s}}(k)$ steps where the polynomial $L_{\mathsf{A_s}}(k)$ bounds the number of steps $\mathsf{A_s}$ can perform.

Putting it all together, H_1 and $\mathsf{A_s}$ simply forward every message between the system Sys and the original adversary A which is represented as a blackbox submachine of the newly defined adversary $\mathsf{A_s}$. Thus, we obtain identical views of the original adversary A, the system Sys, and the honest user H in both configurations. To prove this more formally we could simply go through all possible cases of outputs of A, H, and machines of the system and show that we obtain identical behaviours with respect to the original machines H, A, and the machines of the system in both configurations. We omit it here because it is a rather simple but tedious proof, and we believe that it is already clear by construction of H_1 and $\mathsf{A_s}$ and our above explanations.

As a direct consequence we obtain that the probability of the runs restricted to S does not change, because H_1 always outputs exactly the same tuple to the specified ports as the original A and the view of all machines of the system and the view of H is identical in both configurations. We now combine H and H_1 into one machine $\mathsf{H_s}$. Because of Lemma 2.1 this combination is well-defined and yields a closed collection $\hat{M} \cup \{\mathsf{H_s}, \mathsf{A_s}\}$ again. Moreover, if $conf$ is polynomial-time, H and A are polynomial-time by precondition which implies that $\mathsf{A_s}$ and H_1 are polynomial-time as shown above. Using, Lemma 2.1 we know that $\mathsf{H_s}$ also has to be polynomial-time yielding a polynomial-time configuration

$$conf_{\mathsf{s}} = (\hat{M}, S, \mathsf{H_s}, \mathsf{A_s}) \in \mathsf{Conf}(Sys)$$

in this case. The view of any set of submachines of $\mathsf{H_s}$ and the probability of the runs restricted to S does not change at combination of machines, which yields

$$view_{conf}(\mathsf{H}) = view_{conf_{\mathsf{s}}}(\mathsf{H}) \text{ and } run_{conf} \lceil_S = run_{conf_{\mathsf{s}}} \lceil_S.$$

Finally, $S^c \subseteq \mathsf{ports}(\mathsf{H_s})$ holds by construction, so we have $conf_{\mathsf{s}} \in \mathsf{Conf_s}(Sys)$ which finishes our proof. ∎

A.3 From Section 3.4

Proof. (Theorem 3.5) We first reverse our function φ on the structure $(\varphi(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{X}_{\mathsf{sync},\kappa}\}, S)$ and on the user $\varphi(\mathsf{H}_{\mathsf{sync}})$ yielding the structure $(\hat{M}_{\mathsf{sync}}, S)$ of $Sys_{\mathsf{sync},2}$ and the original honest user $\mathsf{H}_{\mathsf{sync}}$. Note, that we cannot reverse the function φ on the new adversary $\mathsf{A}_{\mathsf{async}}$ in the same way, because we did not demand it to have a similar internal structure, so we construct a new adversary $\mathsf{A}_{\mathsf{sync}}$ for the synchronous configuration as follows. The ports of $\mathsf{A}_{\mathsf{sync}}$ are given by

$$\{\mathsf{p} \mid \mathsf{p}^C \in (\mathsf{ports}(\hat{M}_{\mathsf{sync}}) \cup \mathsf{ports}(\mathsf{H}_{\mathsf{sync}})) \wedge \mathsf{p} \notin (\mathsf{ports}(\hat{M}_{\mathsf{sync}}) \cup \mathsf{ports}(\mathsf{H}_{\mathsf{sync}}))\},$$

i.e., it connects to all remaining free ports of $\hat{M}_{\mathsf{sync}}$ and $\mathsf{H}_{\mathsf{sync}}$. Internally, $\mathsf{A}_{\mathsf{sync}}$ maintains an array $(output\_store_{\mathsf{p}!})_{\mathsf{p}! \in \mathsf{out}(\mathsf{ports}(\mathsf{A}_{\mathsf{async}}))}$ of lists over Σ^* all initially empty.

$\mathsf{A}_{\mathsf{sync}}$ has the adversary $\mathsf{A}_{\mathsf{async}}$ as a blackbox submachine and its behaviour is defined as follows. If $\mathsf{A}_{\mathsf{sync}}$ is clocked in the synchronous system, it gets an input tuple $\mathcal{I} = (\mathcal{I}_{\mathsf{p}?})_{\mathsf{p}? \in \mathsf{in}(\mathsf{ports}(\mathsf{A}_{\mathsf{async}}))}$. It now tries to restore the order of the ports, these messages would have arrived in the asynchronous system. More precisely, it knows the clocking

scheme κ, so it know which machines have been clocking after the last clocking of $\mathsf{A}_{\mathsf{sync}}$. Moreover, it knows the order in which machines are switched by $\mathsf{X}_{\mathsf{sync},\kappa}$ in one particular subround. Using the order on the ports of the asynchronous machines, it can finally decide in which order messages sent by one machine on different ports would have arrived in the asynchronous system. The only problem which might arise is that a machine has been clocked more then once since the last clocking of the adversary. This might result in two inputs at the same port of $\mathsf{A}_{\mathsf{sync}}$ which would be concatenated without any separation symbol. Such an input would not be restorable into its original form, so we had to include the restriction to the considered clocking scheme that every machine and the user are at most clocked once between two successive clockings of the adversary. Note, that our usually used clocking scheme $(\hat{M} \cup \{\mathsf{H}\}, \{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{A}\})$ fulfills this requirement.

After restoring both the usual messages and their order, $\mathsf{A}_{\mathsf{sync}}$ uses the blackbox function $\delta_{\mathsf{A}_{\mathsf{async}}}$ on the first input yielding an output tuple $\mathcal{O}$. This tuple $\mathcal{O}$ is appended to the array $output\_store$, i.e. each component $\mathcal{O}_{\mathsf{p}!}$ is appended to $output\_store_{\mathsf{p}!}$. If there is a nonempty output c at a clockout port $\mathsf{p}^{\triangleleft}!$, we would have a clocked self-loop in $conf_{\mathsf{async}}$ if $output\_store_{\mathsf{p}!}[c] \neq \epsilon$. In this case, this component is removed from the array and $\delta_{\mathsf{A}_{\mathsf{async}}}$ is called again with the new state and $\mathcal{I} := \mathcal{I}_{\mathsf{p}?=output\_store_{\mathsf{p}!}[c]}$ and so on.

Remark A.1. Note, that the adversary $\mathsf{A}_{\mathsf{sync}}$ does not always have to be validly defined if $\mathsf{A}_{\mathsf{async}}$ performs infinite successive clocked self-loops. In this case, it may either diverge or it may yield an infinite probability distribution over possible states and outputs. Therefore, we included the precondition that at least one of the three mentioned modifications has to be made. Obviously, every such modification solves the problem, because a bounded adversary cannot perform infinite self-loops and the third possibility would result in a validly defined adversary $\mathsf{A}_{\mathsf{sync}}$ even in case of such infinite self-loops. ∘

The above steps are repeated with the second input and the new state of $\mathsf{A}_{\mathsf{async}}$ and so on until all inputs have been considered. Finally, the blackbox function is used with $\mathcal{I}_{\mathsf{p}_{\mathsf{A}_{\mathsf{async}}}?=(i,j)}$ where i denotes the global round and j denotes the subround the adversary is clocked in.[4] This correspond to the clocking signal of $\mathsf{X}_{\mathsf{sync},\kappa}$ in the asynchronous system. The output tuple is again concatenated to the same array and possible clocked self loops are considered again. Finally, $\mathsf{A}_{\mathsf{sync}}$ outputs the first elements of each list of $output\_store_{\mathsf{p}!}$ with $\mathsf{p}!^{C} \in \mathsf{ports}(\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}\})$ as its output tuple $\mathcal{O}$ and removes these elements from the lists.

Note, that this newly defined adversary $\mathsf{A}_{\mathsf{sync}}$ is polynomial iff $\mathsf{A}_{\mathsf{async}}$ is polynomial by construction. Thus, if the original configuration $conf_{\mathsf{async}}$ has been polynomial-time (i.e., the user $\varphi(\mathsf{H}_{\mathsf{sync}})$ and the adversary $\mathsf{A}_{\mathsf{async}}$ must be polynomial-time) then the configuration $conf_{\mathsf{sync}} = (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}})$ will also be polynomial-time, since the runtime of $\mathsf{H}_{\mathsf{sync}}$ is always bounded by $\varphi(\mathsf{H}_{\mathsf{sync}})$.

$\mathsf{A}_{\mathsf{sync}}$ "reverse" the function φ by construction. The asynchronous adversary would receive many single inputs, and it would produce outputs every time which would be stored in the outgoing buffers. Possible clocked self-loops are handled by repeated calls of the transition function with correct inputs. If $\mathsf{A}_{\mathsf{async}}$ is scheduled by $\mathsf{X}_{\mathsf{sync},\kappa}$ it again performs an arbitrary transition and the first element of its outgoing buffer would be clocked. The synchronous adversary first splits its input messages into their original

[4] The adversary obviously knows both i and j because he knows the clocking scheme κ, so he may simply maintain two counters that he adapt every time he is clocked.

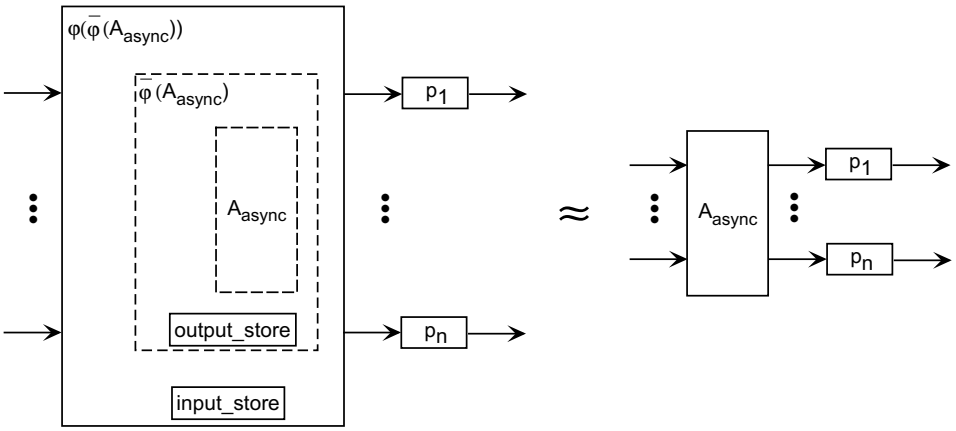

Figure A.1: Overview of the proof of Lemma A.1.

order and uses the blackbox function one by one storing the outputs in $output\_store$. The split inputs correspond to the original inputs of the asynchronous system, so the output tuples are also equal after every step. Therefore, the contents of $output\_store$ always correspond to the outgoing buffers in the asynchronous system after a clocking step of $\mathsf{A}_{\mathsf{async}}$. If the synchronous adversary is clocked it again calls its blackbox function with the correct input and stores the output in the array. After that, it outputs the first element of each list of the array and removes these elements from the lists. In the asynchronous system messages stored in the outgoing buffers are treated in the same way. More formally we could show the following lemma.

Lemma A.1 *We denote this "reversion" of φ_{M} by $\bar{\varphi}_{\mathsf{M}}$ and the reversion of the whole configuration by $\bar{\varphi}_{conf}$ for the moment. Then for an arbitrary configuration $conf_{\mathsf{async}} = (\varphi(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{X}_{\mathsf{sync},\kappa}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \mathsf{A}_{\mathsf{async}})$ we have*

$$view_{\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))}(\varphi(\mathsf{M})) = view_{conf_{\mathsf{async}}}(\varphi(\mathsf{M}))$$

for every $\mathsf{M} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}\})$ and

$$view_{\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))}(\mathsf{A}_{\mathsf{async}}) = view_{conf_{\mathsf{async}}}(\mathsf{A}_{\mathsf{async}})$$

where the view of $\mathsf{A}_{\mathsf{async}}$ in the first configuration is given as a submachine of $\varphi_{\mathsf{M}}(\bar{\varphi}_{\mathsf{M}}(\mathsf{A}_{\mathsf{async}}))$. □

Proof. The proof is illustrated in Figure A.1. We first show that $\mathsf{A}'_{\mathsf{async}} := \varphi_{\mathsf{M}}(\bar{\varphi}_{\mathsf{M}}(\mathsf{A}_{\mathsf{async}}))$ behaves exactly as $\mathsf{A}_{\mathsf{async}}$, i.e., both machines are perfectly indistinguishable for their environment. This is already sufficient to show that the views of $\varphi(\mathsf{M})$ for every $\mathsf{M} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}\})$ are equal in both configurations because they remain unchanged. We will also show that the view of $\mathsf{A}_{\mathsf{async}}$ is equal in both configurations which finishes our proof.

We show that both adversaries $\mathsf{A}'_{\mathsf{async}}$ and $\mathsf{A}_{\mathsf{async}}$ behave identically between two successive clockings. Moreover, we show that the content of array $output\_store_{\mathsf{p!}}$ of $\mathsf{A}'_{\mathsf{async}}$ always equal the outgoing buffers $\widetilde{\mathsf{p}}$ in the corresponding asynchronous configuration at every clocking of $\mathsf{A}_{\mathsf{async}}$ as a submachine of $\mathsf{A}'_{\mathsf{async}}$ if we identify clockings of

$\mathsf{A}_{\mathsf{async}}$ in both configurations in the natural way.[5] Furthermore, we show that outputs made by the adversary are always equal in both configurations.

At the start of the run both buffers and arrays are empty which fulfills our claim. Now assume that $\mathsf{A}'_{\mathsf{async}}$ receives an arbitrary input at $\mathsf{p}? \neq \mathsf{p}_{\mathsf{A}}?$. It stores the message in its array $input\_store_{\mathsf{p}?}$ and gives the control to the master scheduler. If $\mathsf{A}'_{\mathsf{async}}$ receives a non-empty input at $\mathsf{p}_{\mathsf{A}}?$ it applies the state transition function $\delta_{\bar{\varphi}_{\mathsf{M}}(\mathsf{A}_{\mathsf{async}})}$ on the arrays $input\_store$. Now, the arrays $input\_store$ are decomposed into single inputs again preserving their original order, and the function $\delta_{\mathsf{A}_{\mathsf{async}}}$ is applied on every such input. Since the inputs are obviously equal in both configuration, we obtain identical outputs, and moreover identical views for $\mathsf{A}_{\mathsf{async}}$. By precondition, the arrays $output\_store$ are mapped to the outgoing buffers. After one call of $\delta_{\mathsf{A}_{\mathsf{async}}}$, every output at $\mathsf{p}!$ is stored either in $output\_store_{\mathsf{p}!}$ or in $\tilde{\mathsf{p}}$ at the same position, so they remain validly mapped. Now, either the first component of $output\_store_{\mathsf{p}!}$ or the first entry of $\tilde{\mathsf{p}}$ for $\mathsf{p}!^C \in (\mathsf{ports}(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{H}_{\mathsf{sync}}\})$ are output yielding identical outputs and therefore identical views for the environment in both configurations, i.e.,

$$view_{\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))}(\varphi(\mathsf{M})) = view_{conf_{\mathsf{async}}}(\varphi(\mathsf{M}))$$

for $\mathsf{M} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}\})$. We already showed that the views of $\mathsf{A}_{\mathsf{async}}$ are equal in both configurations which finishes our proof. ∎

According to Lemma A.1, the function $\varphi_{conf} \circ \bar{\varphi}_{conf}$ yields identical views for $\varphi(\mathsf{M})$ for every $\mathsf{M} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}\})$ and the asynchronous adversary, i.e.,

- $view_{\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))}(\varphi(\mathsf{M})) = view_{conf_{\mathsf{async}}}(\varphi(\mathsf{M}))$ and
- $view_{\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))}(\mathsf{A}_{\mathsf{async}}) = view_{conf_{\mathsf{async}}}(\mathsf{A}_{\mathsf{async}})$.

We already showed in Theorem 3.4 that $view_{conf_{\mathsf{sync}}}(\mathsf{M}) = \phi(view_{\varphi(conf_{\mathsf{sync}})}(\varphi(\mathsf{M})))$ holds for every synchronous configuration $conf_{\mathsf{sync}} = (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}})$ and for every machine $\mathsf{M} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}}\})$. If we now set $conf_{\mathsf{sync}} := \bar{\varphi}_{conf}(conf_{\mathsf{async}})$, we obtain

- $view_{conf_{\mathsf{sync}}}(\mathsf{M}) = \phi(view_{\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))}(\varphi(\mathsf{M})))$

Moreover, this implies

- $view_{conf_{\mathsf{sync}}}(\mathsf{A}_{\mathsf{sync}}) = \phi(view_{\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))}(\mathsf{A}_{\mathsf{async}}))$

since the views of $\mathsf{A}_{\mathsf{async}}$ and $\varphi(\bar{\varphi}(\mathsf{A}_{\mathsf{async}}))$ are identical. We apply the mapping ϕ on the first two equations and, using Lemma 2.4, we obtain

- $\phi(view_{\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))}(\varphi(\mathsf{M}))) = \phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{M})))$ and
- $\phi(view_{\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))}(\mathsf{A}_{\mathsf{async}})) = \phi(view_{conf_{\mathsf{async}}}(\mathsf{A}_{\mathsf{async}}))$

Note, that ϕ is in fact defined on runs of these configuration because both the machines of the structure and the honest user have the prescribed form. Using transitivity, we immediately obtain the desired result

$$view_{conf_{\mathsf{sync}}}(\mathsf{M}) = \phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{M})))$$

[5] More precisely, this means that we identify the i-th clocking of $\mathsf{A}_{\mathsf{async}}$ in $conf_{\mathsf{async}}$ with the i-th call of $\delta_{\mathsf{A}_{\mathsf{async}}}$ by $\mathsf{A}'_{\mathsf{async}}$ in $\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))$.

and

$$view_{conf_{\mathsf{sync}}}(\mathsf{A}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}}}(\mathsf{A}_{\mathsf{async}}))$$

As a special case we set $\mathsf{M} := \mathsf{H}_{\mathsf{sync}}$ which yields

$$view_{conf_{\mathsf{sync}}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$

■

A.4 From Section 5.3

Recall, that we already gave a transcript of the proof of transitivity of sublists in Section 5.3. For the sake of completeness, and to compare our transcript to our actual PVS proof, we now give the PVS proof in full detail.

Verbose proof for `sublist_transitiv`.
`sublist_transitiv:`

|
|---|
| {1} ∀ (i, j, k: list[T]): sublist(i, j) ∧ sublist(j, k) ⊃ sublist(i, k) |

Inducting on k on formula 1,
we get 2 subgoals:
`sublist_transitiv.1:`

|
|---|
| {1} ∀ (i, j): sublist(i, j) ∧ sublist(j, null) ⊃ sublist(i, null) |

Repeatedly Skolemizing and flattening,
`sublist_transitiv.1:`

{-1} sublist(i′, j′)
{-2} sublist(j′, null)
{1} sublist(i′, null)

Expanding the definition of sublist,
`sublist_transitiv.1:`

{-1} null?(i′) ∨ cons?(j′) ∧ ((car(i′) = car(j′) ∧ sublist(cdr(i′), cdr(j′))) ∨ (sublist(i′, cdr(j′))))
{-2} null?(j′)
{1} null?(i′)

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `sublist_transitiv.1`.

`sublist_transitiv.2:`

```
|
|-------------------------------------------------------------------
| {1}  ∀ (cons1_var: T, cons2_var: list[T]):
|        (∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var)
|                     ⊃ sublist(i, cons2_var))
|        ⊃(∀ (i, j):
|           sublist(i, j) ∧ sublist(j, cons(cons1_var, cons2_var))
|             ⊃ sublist(i, cons(cons1_var, cons2_var)))
```

Repeatedly Skolemizing and flattening,
`sublist_transitiv.2:`

```
| {-1}  ∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var')
|                   ⊃ sublist(i, cons2_var')
| {-2}  sublist(i', j')
| {-3}  sublist(j', cons(cons1_var', cons2_var'))
|-------------------------------------------------------------------
| {1}  sublist(i', cons(cons1_var', cons2_var'))
```

Expanding the definition of sublist,
`sublist_transitiv.2:`

```
| {-1}  ∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var')
|                   ⊃ sublist(i, cons2_var')
| {-2}  sublist(i', j')
| {-3}  sublist(j', cons(cons1_var', cons2_var'))
|-------------------------------------------------------------------
| {1}  null?(i') ∨
|        ((car(i') = cons1_var' ∧
|           sublist(cdr(i'), cons2_var')) ∨
|          (sublist(i', cons2_var')))
```

Applying propositional simplification and decision procedures,
we get 2 subgoals:
`sublist_transitiv.2.1:`

```
| {-1}  ∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var')
|                   ⊃ sublist(i, cons2_var')
| {-2}  sublist(i', j')
| {-3}  sublist(j', cons(cons1_var', cons2_var'))
|-------------------------------------------------------------------
| {1}  car(i') = cons1_var'
| {2}  null?(i')
| {3}  (sublist(i', cons2_var'))
```

Instantiating (with copying) the top quantifier in -1 with the terms: i', j',
`sublist_transitiv.2.1:`

```
| {-1}  ∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var')
|                   ⊃ sublist(i, cons2_var')
| {-2}  sublist(i', j') ∧ sublist(j', cons2_var')
|          ⊃ sublist(i', cons2_var')
| {-3}  sublist(i', j')
| {-4}  sublist(j', cons(cons1_var', cons2_var'))
|-------------------------------------------------------------------
| {1}  car(i') = cons1_var'
| {2}  null?(i')
| {3}  (sublist(i', cons2_var'))
```

Simplifying, rewriting, and recording with decision procedures,
`sublist_transitiv.2.1:`

```
{-1}  ∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var′)
                  ⊃ sublist(i, cons2_var′)
{-2}  sublist(i′, j′)
{-3}  sublist(j′, cons(cons1_var′, cons2_var′))
------------------------------------------------
{1}   car(i′) = cons1_var′
{2}   sublist(j′, cons2_var′)
{3}   null?(i′)
{4}   (sublist(i′, cons2_var′))
```

Expanding the definition of sublist,
`sublist_transitiv.2.1:`

```
{-1}  ∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var′)
                  ⊃ sublist(i, cons2_var′)
{-2}  sublist(i′, j′)
{-3}  null?(j′) ∨
        (car(j′) = cons1_var′ ∧
         sublist(cdr(j′), cons2_var′))
------------------------------------------------
{1}   car(i′) = cons1_var′
{2}   sublist(j′, cons2_var′)
{3}   null?(i′)
{4}   (sublist(i′, cons2_var′))
```

Applying propositional simplification and decision procedures,
we get 2 subgoals:
`sublist_transitiv.2.1.1:`

```
{-1}  null?(j′)
{-2}  ∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var′)
                  ⊃ sublist(i, cons2_var′)
{-3}  sublist(i′, j′)
------------------------------------------------
{1}   car(i′) = cons1_var′
{2}   sublist(j′, cons2_var′)
{3}   null?(i′)
{4}   (sublist(i′, cons2_var′))
```

Expanding the definition of sublist,

`sublist_transitiv.2.1.1:`

{-1} null?(j')
{-2} $\forall$ (i, j):
(null?(i) $\vee$ cons?(j) $\wedge$
((car(i) = car(j) $\wedge$
sublist(cdr(i), cdr(j))) $\vee$
(sublist(i, cdr(j))))) $\wedge$
(null?(j) $\vee$
((car(j) = car(cons2_var$'$) $\wedge$
sublist(cdr(j), cdr(cons2_var$'$))) $\vee$
(sublist(j, cdr(cons2_var$'$)))))
$\supset$
null?(i) $\vee$
((car(i) = car(cons2_var$'$) $\wedge$
sublist(cdr(i), cdr(cons2_var$'$))) $\vee$
(sublist(i, cdr(cons2_var$'$))))
{-3} FALSE

{1} car(i') = cons1_var$'$
{2} TRUE
{3} null?(i')
{4} ((car(i') = car(cons2_var$'$) $\wedge$
sublist(cdr(i'), cdr(cons2_var$'$))) $\vee$
(sublist(i', cdr(cons2_var$'$))))

which is trivially true.
This completes the proof of `sublist_transitiv.2.1.1`.
`sublist_transitiv.2.1.2:`

{-1} car(j') = cons1_var$'$
{-2} sublist(cdr(j'), cons2_var$'$)
{-3} $\forall$ (i, j): sublist(i, j) $\wedge$ sublist(j, cons2_var$'$)
$\supset$ sublist(i, cons2_var$'$)
{-4} sublist(i', j')

{1} car(i') = cons1_var$'$
{2} sublist(j', cons2_var$'$)
{3} null?(i')
{4} (sublist(i', cons2_var$'$))

Instantiating the top quantifier in -3 with the terms: i', cdr(j'),
we get 2 subgoals:
`sublist_transitiv.2.1.2.1:`

{-1} car(j') = cons1_var$'$
{-2} sublist(cdr(j'), cons2_var$'$)
{-3} sublist(i', cdr(j')) $\wedge$ sublist(cdr(j'), cons2_var$'$)
$\supset$ sublist(i', cons2_var$'$)
{-4} sublist(i', j')

{1} car(i') = cons1_var$'$
{2} sublist(j', cons2_var$'$)
{3} null?(i')
{4} (sublist(i', cons2_var$'$))

Simplifying, rewriting, and recording with decision procedures,
`sublist_transitiv.2.1.2.1:`

{-1}	car(j') = cons1_var$'$
{-2}	sublist(cdr(j'), cons2_var$'$)
{-3}	sublist(i', j')
{1}	car(i') = cons1_var$'$
{2}	sublist(i', cdr(j'))
{3}	sublist(j', cons2_var$'$)
{4}	null?(i')
{5}	(sublist(i', cons2_var$'$))

Expanding the definition of sublist,
`sublist_transitiv.2.1.2.1:`

{-1}	car(j') = cons1_var$'$
{-2}	sublist(cdr(j'), cons2_var$'$)
{-3}	FALSE
{1}	car(i') = cons1_var$'$
{2}	sublist(i', cdr(j'))
{3}	sublist(j', cons2_var$'$)
{4}	null?(i')
{5}	(sublist(i', cons2_var$'$))

which is trivially true.
This completes the proof of `sublist_transitiv.2.1.2.1`.
`sublist_transitiv.2.1.2.2:`

{-1}	car(j') = cons1_var$'$
{-2}	sublist(cdr(j'), cons2_var$'$)
{-3}	sublist(i', j')
{1}	cons?[T](j')
{2}	car(i') = cons1_var$'$
{3}	sublist(j', cons2_var$'$)
{4}	null?(i')
{5}	(sublist(i', cons2_var$'$))

Expanding the definition of sublist,

`sublist_transitiv.2.1.2.2:`

```
{-1}  car(j') = cons1_var'
{-2}  null?(cdr(j')) ∨
        cons?(cons2_var') ∧
          ((car(cdr(j')) = car(cons2_var') ∧
              sublist(cdr(cdr(j')), cdr(cons2_var'))) ∨
            (sublist(cdr(j'), cdr(cons2_var'))))
{-3}  cons?(j') ∧ (sublist(i', cdr(j')))
------------------------------------------------------
{1}   cons?[T](j')
{2}   car(i') = cons1_var'
{3}   null?(j') ∨
        cons?(cons2_var') ∧
          ((car(j') = car(cons2_var') ∧
              sublist(cdr(j'), cdr(cons2_var'))) ∨
            (sublist(j', cdr(cons2_var'))))
{4}   null?(i')
{5}   cons?(cons2_var') ∧
        ((car(i') = car(cons2_var') ∧
            sublist(cdr(i'), cdr(cons2_var'))) ∨
          (sublist(i', cdr(cons2_var'))))
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `sublist_transitiv.2.1.2.2`.
`sublist_transitiv.2.2:`

```
{-1}  ∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var')
                  ⊃ sublist(i, cons2_var')
{-2}  sublist(i', j')
{-3}  sublist(j', cons(cons1_var', cons2_var'))
------------------------------------------------------
{1}   sublist(cdr(i'), cons2_var')
{2}   null?(i')
{3}   (sublist(i', cons2_var'))
```

Expanding the definition of sublist,
`sublist_transitiv.2.2:`

```
{-1}  ∀ (i, j): sublist(i, j) ∧ sublist(j, cons2_var')
                  ⊃ sublist(i, cons2_var')
{-2}  sublist(i', j')
{-3}  null?(j') ∨
        ((car(j') = cons1_var' ∧
            sublist(cdr(j'), cons2_var')) ∨
          (sublist(j', cons2_var')))
------------------------------------------------------
{1}   sublist(cdr(i'), cons2_var')
{2}   null?(i')
{3}   (sublist(i', cons2_var'))
```

Applying propositional simplification and decision procedures,
we get 3 subgoals:

`sublist_transitiv.2.2.1:`

{-1} null?(j')
{-2} $\forall$ (i, j): sublist(i, j) $\wedge$ sublist(j, cons2_var$'$) $\supset$ sublist(i, cons2_var$'$)
{-3} sublist(i', j')

{1} sublist(cdr(i'), cons2_var$'$)
{2} null?(i')
{3} (sublist(i', cons2_var$'$))

Expanding the definition of sublist,
`sublist_transitiv.2.2.1:`

{-1} null?(j')
{-2} $\forall$ (i, j):
(null?(i) $\vee$ cons?(j) $\wedge$
((car(i) = car(j) $\wedge$
sublist(cdr(i), cdr(j))) $\vee$
(sublist(i, cdr(j)))) $\wedge$
(null?(j) $\vee$
((car(j) = car(cons2_var$'$) $\wedge$
sublist(cdr(j), cdr(cons2_var$'$))) $\vee$
(sublist(j, cdr(cons2_var$'$)))))
$\supset$
null?(i) $\vee$
((car(i) = car(cons2_var$'$) $\wedge$
sublist(cdr(i), cdr(cons2_var$'$))) $\vee$
(sublist(i, cdr(cons2_var$'$))))
{-3} FALSE

{1} null?(cdr(i')) $\vee$
((car(cdr(i')) = car(cons2_var$'$) $\wedge$
sublist(cdr(cdr(i')), cdr(cons2_var$'$))) $\vee$
(sublist(cdr(i'), cdr(cons2_var$'$))))
{2} null?(i')
{3} ((car(i') = car(cons2_var$'$) $\wedge$
sublist(cdr(i'), cdr(cons2_var$'$))) $\vee$
(sublist(i', cdr(cons2_var$'$))))

which is trivially true.
This completes the proof of `sublist_transitiv.2.2.1`.
`sublist_transitiv.2.2.2:`

{-1} car(j') = cons1_var$'$
{-2} sublist(cdr(j'), cons2_var$'$)
{-3} $\forall$ (i, j): sublist(i, j) $\wedge$ sublist(j, cons2_var$'$) $\supset$ sublist(i, cons2_var$'$)
{-4} sublist(i', j')

{1} sublist(cdr(i'), cons2_var$'$)
{2} null?(i')
{3} (sublist(i', cons2_var$'$))

Instantiating the top quantifier in -3 with the terms: cdr(i'), cdr(j'),
we get 2 subgoals:

`sublist_transitiv.2.2.2.1:`

{-1}	car(j') = cons1_var′
{-2}	sublist(cdr(j'), cons2_var′)
{-3}	sublist(cdr(i'), cdr(j')) ∧ sublist(cdr(j'), cons2_var′) ⊃ sublist(cdr(i'), cons2_var′)
{-4}	sublist(i', j')
{1}	sublist(cdr(i'), cons2_var′)
{2}	null?(i')
{3}	(sublist(i', cons2_var′))

Simplifying, rewriting, and recording with decision procedures,
`sublist_transitiv.2.2.2.1:`

{-1}	car(j') = cons1_var′
{-2}	sublist(cdr(j'), cons2_var′)
{-3}	sublist(i', j')
{1}	sublist(cdr(i'), cons2_var′)
{2}	sublist(cdr(i'), cdr(j'))
{3}	null?(i')
{4}	(sublist(i', cons2_var′))

Using lemma sublist_lem_1,
`sublist_transitiv.2.2.2.1:`

{-1}	sublist(i', j') ∧ cons?(i') ∧ cons?(j') ⊃ sublist(cdr(i'), cdr(j'))
{-2}	car(j') = cons1_var′
{-3}	sublist(cdr(j'), cons2_var′)
{-4}	sublist(i', j')
{1}	sublist(cdr(i'), cons2_var′)
{2}	sublist(cdr(i'), cdr(j'))
{3}	null?(i')
{4}	(sublist(i', cons2_var′))

Simplifying, rewriting, and recording with decision procedures,
`sublist_transitiv.2.2.2.1:`

{-1}	car(j') = cons1_var′
{-2}	sublist(cdr(j'), cons2_var′)
{-3}	sublist(i', j')
{1}	cons?(j')
{2}	sublist(cdr(i'), cons2_var′)
{3}	sublist(cdr(i'), cdr(j'))
{4}	null?(i')
{5}	(sublist(i', cons2_var′))

Expanding the definition of sublist,

`sublist_transitiv.2.2.2.1:`

{-1}	car(j') = cons1_var′
{-2}	null?(cdr(j')) ∨ cons?(cons2_var′) ∧ ((car(cdr(j')) = car(cons2_var′) ∧ sublist(cdr(cdr(j')), cdr(cons2_var′))) ∨ (sublist(cdr(j'), cdr(cons2_var′))))
{-3}	FALSE
{1}	cons?(j')
{2}	null?(cdr(i')) ∨ cons?(cons2_var′) ∧ ((car(cdr(i')) = car(cons2_var′) ∧ sublist(cdr(cdr(i')), cdr(cons2_var′))) ∨ (sublist(cdr(i'), cdr(cons2_var′))))
{3}	null?(cdr(i')) ∨ cons?(cdr(j')) ∧ ((car(cdr(i')) = car(cdr(j')) ∧ sublist(cdr(cdr(i')), cdr(cdr(j')))) ∨ (sublist(cdr(i'), cdr(cdr(j')))))
{4}	null?(i')
{5}	cons?(cons2_var′) ∧ ((car(i') = car(cons2_var′) ∧ sublist(cdr(i'), cdr(cons2_var′))) ∨ (sublist(i', cdr(cons2_var′))))

which is trivially true.
This completes the proof of `sublist_transitiv.2.2.2.1`.
`sublist_transitiv.2.2.2.2:`

{-1}	car(j') = cons1_var′
{-2}	sublist(cdr(j'), cons2_var′)
{-3}	sublist(i', j')
{1}	cons?[T](j')
{2}	sublist(cdr(i'), cons2_var′)
{3}	null?(i')
{4}	(sublist(i', cons2_var′))

Expanding the definition of sublist,

`sublist_transitiv.2.2.2.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{car}(j') = \text{cons1\_var}' \\
\{\text{-}2\} & \text{null?}(\text{cdr}(j')) \vee \\
 & \quad \text{cons?}(\text{cons2\_var}') \wedge \\
 & \quad\quad ((\text{car}(\text{cdr}(j')) = \text{car}(\text{cons2\_var}') \wedge \\
 & \quad\quad\quad \text{sublist}(\text{cdr}(\text{cdr}(j')), \text{cdr}(\text{cons2\_var}'))) \vee \\
 & \quad\quad\quad (\text{sublist}(\text{cdr}(j'), \text{cdr}(\text{cons2\_var}')))) \\
\{\text{-}3\} & \text{cons?}(j') \wedge \\
 & \quad ((\text{car}(i') = \text{car}(j') \wedge \\
 & \quad\quad \text{sublist}(\text{cdr}(i'), \text{cdr}(j'))) \vee \\
 & \quad\quad (\text{sublist}(i', \text{cdr}(j')))) \\
\hline
\{1\} & \text{cons?}[T](j') \\
\{2\} & \text{null?}(\text{cdr}(i')) \vee \\
 & \quad \text{cons?}(\text{cons2\_var}') \wedge \\
 & \quad\quad ((\text{car}(\text{cdr}(i')) = \text{car}(\text{cons2\_var}') \wedge \\
 & \quad\quad\quad \text{sublist}(\text{cdr}(\text{cdr}(i')), \text{cdr}(\text{cons2\_var}'))) \vee \\
 & \quad\quad\quad (\text{sublist}(\text{cdr}(i'), \text{cdr}(\text{cons2\_var}')))) \\
\{3\} & \text{null?}(i') \\
\{4\} & \text{cons?}(\text{cons2\_var}') \wedge \\
 & \quad ((\text{car}(i') = \text{car}(\text{cons2\_var}') \wedge \\
 & \quad\quad \text{sublist}(\text{cdr}(i'), \text{cdr}(\text{cons2\_var}'))) \vee \\
 & \quad\quad (\text{sublist}(i', \text{cdr}(\text{cons2\_var}'))))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `sublist_transitiv.2.2.2.2`.
`sublist_transitiv.2.2.3:`

$$
\begin{array}{ll}
\{\text{-}1\} & (\text{sublist}(j', \text{cons2\_var}')) \\
\{\text{-}2\} & \forall\ (i, j):\ \text{sublist}(i, j) \wedge \text{sublist}(j, \text{cons2\_var}') \\
 & \quad\quad \supset \text{sublist}(i, \text{cons2\_var}') \\
\{\text{-}3\} & \text{sublist}(i', j') \\
\hline
\{1\} & \text{sublist}(\text{cdr}(i'), \text{cons2\_var}') \\
\{2\} & \text{null?}(i') \\
\{3\} & (\text{sublist}(i', \text{cons2\_var}'))
\end{array}
$$

Instantiating the top quantifier in -2 with the terms: i', j',
`sublist_transitiv.2.2.3:`

$$
\begin{array}{ll}
\{\text{-}1\} & (\text{sublist}(j', \text{cons2\_var}')) \\
\{\text{-}2\} & \text{sublist}(i', j') \wedge \text{sublist}(j', \text{cons2\_var}') \\
 & \quad\quad \supset \text{sublist}(i', \text{cons2\_var}') \\
\{\text{-}3\} & \text{sublist}(i', j') \\
\hline
\{1\} & \text{sublist}(\text{cdr}(i'), \text{cons2\_var}') \\
\{2\} & \text{null?}(i') \\
\{3\} & (\text{sublist}(i', \text{cons2\_var}'))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `sublist_transitiv.2.2.3`.
Q.E.D.

Bibliography

[1] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. 4th Intern. Conf. Foundations of Software Science and Computation Structures (FOSSACS 2001), Springer-Verlag, Berlin 2001, 25-41.

[2] M. Abadi and 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 and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. Information and Computation 148/1 (1999) 1-70.

[4] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). IFIP Intern. Conf. on Theoretical Computer Science (TCS 2000), LNCS 1872, Springer-Verlag, 2000, 3–22.

[5] B. Alpern and F. B. Schneider. Defining liveness. Information Processing Letters 21 (1985) 181-185.

[6] M. Backes. A calculus for probabilistic bisimulation and its usefulness for modern cryptography. Unpublished Manusscript.

[7] D. Beaver and S. Haber. Cryptographic protocols provably secure against dynamic adversaries. Eurocrypt '92, LNCS 658, Springer-Verlag, Berlin 1993, 307-323.

[8] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and multics interpretation. D. Elliott Bell and Leonard. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, The Mitre Corporation, Bedford MA, USA, March 1976.

[9] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. Crypto '98, LNCS 1462, Springer-Verlag, 1998, 26–45.

[10] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. Computer Aided Verification (CAV'94), Springer-Verlag, June 1994, 68-80.

[11] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. Cryptology ePrint Archive, Report 2001/006, Mar. 2001. http://eprint.iacr.org. Full length version of the extended abstract in Proc. Crypto 2001.

[12] R. Canetti. Security and composition of multiparty cryptographic protocols. Journal of Cryptology 13/1 (2000) 143-202.

[13] D. Clark, C. Hankin, S. Hunt, and R. Nagarajan. Possibilistic information flow is safe for probabilistic non-interference. WITS00, available at www.doc.ic.ac.uk/ clh/Papers/witscnh.ps.gz.

[14] R. Cramer and V. Shoup. Practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. Crypto '98, LNCS 1462, Springer-Verlag, 1998, 13–25.

[15] D. E. Denning. A lattice model of secure information flow. Communications of the ACM 19/5 (1976) 236-243.

[16] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. Communications of the ACM 20/7 (1977) 504-513.

[17] D. Dolev and A. C. Yao. On the security of public key protocols. IEEE Transactions on Information Theory 29/2 (1983) 198-208.

[18] F. J. T. Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? 1998 IEEE Symposium on Research in Security and Privacy, Oakland, 1998, 160-171.

[19] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. IEEE Transactions on Software Engineering 23/9 (1997) 550-571.

[20] J. A. Goguen and J. Meseguer. Security policies and security models. IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Washington 1982, 11-20.

[21] J. A. Goguen and J. Meseguer. Unwinding and inference control. IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Oakland 1984, 75-86.

[22] S. Goldwasser and S. Micali. Probabilistic encryption. Journal of Computer and System Sciences 28 (1984), 270-299.

[23] J. W. Gray III. Probabilistic interference. IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos 1990, 170-179.

[24] J. W. Gray III. Toward a mathematical foundation for information flow security. Journal of Computer Science, Vol.1, Num. 3,4, 1992, 255-295.

[25] J. Haigh and W. Young. Extending the non-interference version of mls for sat. IEEE Symposium on Security and Privacy, IEEE Computer Society Press, April 1986, 232-239.

[26] C. A. R. Hoare. Communicating sequential processes. International Series in Computer Science, Prentice Hall, Hemel Hempstead 1985.

[27] M. H. Kang, I. S. Moskowitz, and D. C. Lee. A network version of the pump. IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos 1995, 144-154.

[28] P. Laud. Semantics and program analysis of computationally secure information flow. 10th European Symposium On Programming (ESOP 2001), LNCS 2028, Springer-Verlag, Berlin 2001, 77-91.

[29] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. 5th Conf. on Computer and Communications Security, ACM, 1998, 112–12.

[30] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security analysis. Formal Methods '99, LNCS 1708, Springer-Verlag, 1999, 776–793.

[31] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, 1996, 147-166.

[32] N. Lynch. Distributed algorithms. Morgan Kaufmann Publishers, San Francisco 1996.

[33] N. Lynch. I/o automaton models and proofs for shared-key communication systems. 12th Computer Security Foundations Workshop (CSFW), IEEE, 1999, 14–29.

[34] H. Mantel. Unwinding possibilistic security properties. ESORICS '00 (6th European Symposium on Research in Computer Security), Toulouse, LNCS 1895, Springer-Verlag, Berlin 2000, 238-254.

[35] H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. 14th IEEE Computer Security Foundations Workshop (CSFW'01), IEEE Computer Society Press, Cape Breton 2001, 200-214.

[36] D. McCullough. Specifications for multi-level security and a hook-up property. IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Oakland 1987,, 161-166.

[37] J. McLean. Security models. in: John Marciniak (ed.): Encyclopedia of Software Engineering; Wiley Press, 1994.

[38] J. McLean. Security models and information flow. IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos 1990, 180-187.

[39] C. Meadows. Using narrowing in the analysis of key management protocols. Symposium on Security and Privacy, IEEE, 1989, 138–147.

[40] J. K. Millen. Covert channel capacity. Proc. 1987 IEEE Symp. on Security and Privacy, April 27-29, 1987, Oakland, California, 60-66.

[41] J. K. Millen. The interrogator: A tool for cryptographic protocol security. Symposium on Security and Privacy, IEEE, 1984, 134–141.

[42] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology, 2000, 410-442.

[43] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical report, Computer Science Laboratory, SRI International, 1993.

[44] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *CADE 11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.

[45] L. Paulson. The inductive approach to verifying cryptographic protocols. Journal of Computer Security, 6(1):85-128, 1998.

[46] B. Pfitzmann, M. Schunter, and M. Waidner. Provably secure certified mail. IBM Research Report RZ 3207 (#93253) 02/14/2000, IBM Research Division, Z¨urich, August 2000.

[47] B. Pfitzmann, M. Schunter, and M. Waidner. Secure reactive systems. IBM Research Report RZ 3206 (#93252) 02/14/2000, IBM Research Division, Z¨urich, May 2000.

[48] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. 7th ACM Conference on Computer and Communications Security, Athens, November 2000, ACM Press, New York 2000, 245-254.

[49] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. IEEE Symposium on Security and Privacy, Oakland, May 2001, 184-201.

[50] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. Crypto '91, LNCS 576, Springer-Verlag, Berlin 1992, 433-444.

[51] A. Roscoe and M. Goldreich. What is intransitive noninterference? 12th IEEE Computer Security Foundations Workshop (CSFW'99), IEEE Computer Society Press, 1999, 226-238.

[52] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. 8th Computer Security Foundations Workshop (CSFW'95), IEEE, 1995, 98–107.

[53] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. 8th European Symposium on Programming, ESOP'99, Springer Verlag, 1999, 40-58.

[54] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. 13th Computer Security Foundations Workshop (CSFW'00), IEEE Computer Society Press, 2000, 200-214.

[55] S. Schneider. Security properties and CSP. 1996 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Washington 1996, 174-187.

[56] M. Schunter. Optimistic fair exchange. Dissertation, Technische Fakult¨at, Universit¨at des Saarlandes, 2000.

[57] G. Smith. A new type system for secure information flow. 14th IEEE Computer Security Foundations Workshop (CSFW'01), IEEE Computer Society Press, Cape Breton 2001, 115-125.

[58] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. 25th Symposium on Principles of Programming Languages (POPL), ACM, New York 1998, 355-364.

[59] M. Steiner. Secure group key exchange. Dissertation, Technische Fakult¨at, Universit¨at des Saarlandes, 2001.

[60] D. Sutherland. A model of information. Proceedings of the 9th National Computer Security Conference; National Bureau of Standards, National Computer Security Center, 15.-18. September 1986, 175-183.

[61] D. Volpano. Secure introduction of one-way functions. 13th Computer Security Foundations Workshop, IEEE Computer Society Press, Los Alamitos 2000, 246-254.

[62] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. 10th Computer Security Foundations Workshop (CSFW'97), IEEE Computer Society Press, Los Alamitos 1997, 156-168.

[63] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. 11th Computer Security Foundations Workshop (CSFW'98), IEEE Computer Society Press, Los Alamitos 1998, 34-43.

[64] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos 1990, 144-161.

[65] A. C. Yao. Protocols for secure computations. 23rd Symposium on Foundations of Computer Science (FOCS) 1982, IEEE Computer Society, 1982, 160-164.

[66] A. Zakinthinos and E. S. Lee. A general theory of security properties. IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Washington 1997, 94-102.

[67] S. Zdancewic and A. C. Myers. Robust declassification. 14th IEEE Computer Security Foundation Workshop, Cape Breton 2001, 15-23.

Index