Unifying Simulatability Definitions in Cryptographic Systems under Different Timing Assumptions*

Michael Backes

IBM Zurich Research Laboratory, Rüschlikon, Switzerland
mbc@zurich.ibm.com

Abstract. The cryptographic concept of simulatability has become a salient technique for faithfully analyzing and proving security properties of arbitrary cryptographic protocols. We investigate the relationship between simulatability in synchronous and asynchronous frameworks by means of the formal models of Pfitzmann et. al., which are seminal in using this concept in order to bridge the gap between the formal-methods and the cryptographic community. We show that the synchronous model can be seen as a special case of the asynchronous one with respect to simulatability, i.e., we present an embedding between both models that we show to preserve simulatability. We show that this result allows for carrying over lemmas and theorems that rely on simulatability from the asynchronous model to its synchronous counterpart without any additional work. Hence future work can concentrate on the more general asynchronous case, without having to neglect the analysis of synchronous protocols.

1 Introduction

In recent times, the analysis of cryptographic protocols has been getting more and more attention, and the demand for general frameworks for representing cryptographic protocols and the security requirements of cryptographic tasks has been rising. Existing framework are either motivated by the complexity-theoretic view on cryptography, which aims at proving cryptographic protocols with respect to the cryptographic semantics, or they are motivated by the view of the formal-methods community, which aims at capturing abstractions of cryptography in order to make such protocols accessible for formal verification. Frameworks built on abstractions will be further dealt with in the related literature along with a discussion on the cryptographic justification of these abstractions.

For living up to the probabilistic nature of cryptography, a framework for dealing with actual cryptography necessarily has to be able to deal with probabilistic behaviors. The standard understanding in well-known, non security-specific probabilistic frameworks like [38, 41] is that the order of events is fixed by means of a probabilistic scheduler that has full information about the system. In contrast to that, the standard understanding in cryptology (closest to a rigorous definition in [10]) is that the adversary schedules everything, but only with realistic information. This corresponds to making a certain subclass of schedulers explicit for the model from [38]. However, if one splits a machine into local submachines, or defines intermediate systems for the purposes of proof only, this may introduce many schedules that do not correspond to a schedule of the original system and therefore just complicate the proofs. The typical solution is a distributed definition of scheduling which allows machines that have been scheduled to schedule certain (statically fixed) other machines themselves.

Based on these requirements, several general definitions of secure protocols were developed over the years, e.g. [15, 28, 7, 23, 35, 18, 11, 37, 12], which are all potential candidates for such a framework. To allow for a faithful analysis of cryptographic protocols, it is well-known that such models not only have to capture probabilistic behaviors, but also complexity-theoretically bounded adversaries as well as a reactive environment of the protocol, i.e., continuous interaction with the users and the adversary. Unfortunately, most of the above work does not live up to these requirements in spite of its generality, mainly since it

* An extended abstract of this work appears in the 14th international conference on concurrency theory (CONCUR 2003).

concentrates on the task of secure function evaluation, which does not capture a reactive environment. Currently, the models of Pfitzmann et. al. [35, 37] and Canetti [12], which have been developed concurrently but independently, stand out as the standard models for sound protocol analysis and design.

Regarding the underlying definition of time, such models can be split into synchronous and asynchronous ones. In synchronous models [35], time is assumed to be expressible in rounds, whereas asynchronous scenarios [37, 12] do not impose any assumption on time. This makes asynchronous scenarios attractive since no assumption is made about network delays and the relative execution speed of the parties. Moreover, the notion of rounds is difficult to justify in practice as it seems to be very difficult to establish them for the Internet for example. This attractiveness is substantiated by a large body of literature on asynchronous cryptographic protocols, e.g., [8, 14]. However, time guarantees are sometimes explicitly desired, e.g., on when a process can abort. Hence assumptions have to be made in this case, which induce a certain amount of synchrony again. This sometimes makes a synchronous assumption of time nevertheless necessary in practice, e.g., in Kerberos [30].

Hence researchers usually restrict their attention to one definition of time, or they are driving double-tracked by maintaining two separate models. However, this presupposes proving every theorem for both models. This is not nice. An alternative approach, taken in this work, is to show that the synchronous model can be regarded as a special case of an asynchronous one, and hence does not have to be considered separately, but still can be used to conveniently express synchronous protocols.

Although this idea might not be surprising, it is very difficult to achieve since it turns out that carrying over results from the asynchronous to the embedded synchronous model presupposes the possibility of (at least partially) reversing the considered embedding. Recall that suitable frameworks, especially the frameworks of Canetti and Pfitzmann et. al., have a distributed scheduling which significantly complicates this reversion.

Formally, a special case means that there is an embedding into the asynchronous model that preserves a desired property. Which property has to be preserved depends on the goals to strive for. For cryptographic protocols, the property of *simulatability* stands out. Simulatability captures the notion of a cryptographically secure implementation and serves as a link to the formal-methods community, which typically only hold a top-level view of cryptography, where cryptographic primitives are replaced by deterministic abstractions. A more comprehensive discussion of simulatability and its relationship to protocol verification work done by the formal-methods community is given in the paragraph on related literature below.

In the following, we investigate the synchronous and asynchronous models of Pfitzmann et. al. [35, 37], which are seminal in using the concept of simulatability to bridge the gap between the formal-methods and the cryptographic community. We show that the synchronous model can be embedded in the asynchronous model such that simulatability is preserved by this embedding, i.e., if two systems fulfill the simulatability relation in the synchronous model, their respective images fulfill the relation in the asynchronous model and vice versa. We show that this result allows for carrying over lemmas and theorems from the asynchronous case to the synchronous case without proving them twice. Hence future work can concentrate on the more general asynchronous case without neglecting the analysis of synchronous protocols. We are confident that this result helps to make future protocol analysis in these models more convenient and more efficient.

Moreover, we believe that our approach for establishing the embedding and its properties can be successfully used for other models with only minor changes. Especially the asynchronous model of Canetti is surely worth to be investigated. However, his corresponding synchronous model [11] is still specific for secure function evaluation; hence adopting it to a reactive environment is a necessary prerequisite for this future work. The lack of such a reactive synchronous model was – besides the fact that the models of Pfitzmann et. al. are more rigorously defined than the one of Canetti – our main reason why we decided to base our work on the model of Pfitzmann et. al.

Related Literature. If cryptographic protocols should be verified using formal methods, some kind of abstraction is needed as the underlying reduction proofs of cryptography are still out of scope of current verification techniques. This abstraction is usually based on the so-called Dolev-Yao abstraction [13], which considers cryptographic primitives, e.g., E for encryption and D for decryption, as operators in a free algebra where only predefined cancellation rules hold. For instance, twofold encryption of a message m does not yield another message from the basic message space but the term $E(E(m))$. A typical cancellation rule is $D(E(m)) = m$. This abstraction simplifies proofs of larger protocols considerably, and it gave rise to a large body of literature on analyzing the security of protocols using techniques for formal verification of computer programs (a very partial list of work includes [29, 26, 20, 9, 27, 21, 24, 33, 39, 1]).

Since this line of work turned out to be very successful, the interesting question arose whether these abstractions are indeed justified from the view of cryptography, i.e., whether properties proved for the abstractions are still valid for the cryptographic implementation. Abadi et. al. showed in [3, 2] that the Dolev-Yao model is cryptographically faithful at least for symmetric encryption and synchronous protocols. There, however, the adversary is restricted to passive eavesdropping. Consequently, it was not necessary to choose a reactive model of a system and its honest users, and the notion of simulatability could be replaced by the weaker notion of indistinguishability [43]. Another interesting approach has been presented by Guttman et. al. [17] which show that the probability of two executions of the same protocol – either executed in a Dolev-Yao-like framework or using real cryptographic primitives – may deviate from each other at most for a certain bound. However, their results are specific for the Wegman-Carter system so far. Moreover, as this system is information-theoretically secure, its security proof is much easier to handle than primitives with security guarantees only against computationally bounded adversaries since no reduction proofs against underlying number-theoretic assumptions have to be made. Some further approaches for special security goals or primitives are [40, 22]. However, there is evidence that the original Dolev-Yao model is not justified in the presence of active attacks, even if provably secure cryptographic primitives are used, cf. [34] for an (admittedly constructed) counterexample. This exemplifies the demand for "better" abstractions which the models of Canetti and of Pfitzmann et. al. want to establish using the concept of simulatability.

Simulatability bridges this gap by serving as a cryptographically sufficient relationship between abstract specifications and cryptographic implementations, i.e., abstractions which can be shown to simulate a given implementation in a particular sense are known to be sound with respect to the security definitions of cryptography. Simulatability was first invented for multi-party function evaluation [42, 15, 7, 28, 11], i.e., systems with only one initial input set and only one output set. An extension to a reactive scenario, where participants can make new inputs many times, e.g., start new sessions like key exchanges, was first fully defined in [34], with extensions to asynchronous systems in [37, 12]. Each of the three considered models was already successfully used to built up sound abstractions of various cryptographic primitives and all of them enjoy a composition theorem, i.e., large protocols can be refined step-wise without destroying the already proven properties.

Comparing the models of Canetti and Pfitzmann et. al., we can state that Canetti's work enjoys a more general composition theorem and has moreover addressed more cryptographic primitives so far. On the other hand, the models of Pfitzmann et. al. are more rigorously defined and early examples of tool-supported proofs in their models exist [5, 4], using PVS [32]. Moreover, the recently published universally composable cryptographic library [6] may pave the way to formal verification of large security protocols within their models.

Outline. In Section 2 we review the reactive models for synchronous and asynchronous time. In Section 3, we explain how the embedding works and give a rigorous definition. Starting with a proof sketch of the first embedding theorem in Section 4 (there will be two of them) and some lemmas capturing essential steps in the theorem's proof, we fade to the embedding theorems in Section 5. In conjunction, both theorems allow for carrying over theorems from the asynchronous to the synchronous case, which is shown in Section 6 by means of an example. For the sake of readability, most occurring proofs are postponed to the Appendix.

2 Review of the Reactive Models in Synchronous and Asynchronous Networks

In this section we briefly review the synchronous and the asynchronous model for probabilistic reactive systems as introduced in [35] and [37], respectively. Several definitions are only sketched, whereas those that are essential for understanding our upcoming results are given in full detail.

2.1 General System Model

In the following we consider a finite alphabet Σ and 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-letter prefix of s.

Our machine model is *probabilistic state-transition machines*, similar to probabilistic I/O automata as sketched by Lynch [25]. Communication between different machines is done via *ports* which are divided into *input* and *output* ports. Inspired by the CSP-Notation [19] we write input and output ports as $\mathsf{q}?$ and $\mathsf{q}!$.

Ports will later be connected by naming convention, i.e., a port $\mathsf{q}!$ always sends messages to $\mathsf{q}?$. In the asynchronous model, a special machine called a *buffer* will further be inserted in each connection to ensure asynchronous behavior. A buffer stores all of its inputs in an internal list. If a machine wants to schedule the i-th message of buffer $\widetilde{\mathsf{q}}$ (this machine must have the unique clock-out port $\mathsf{q}^{\triangleleft}!$), it simply sends i at $\mathsf{q}^{\triangleleft}!$, cf. Figure 1. The buffer then schedules the i-th message and removes it from its internal list. Neither buffers nor clock ports occur in synchronous machines; they are just included to establish a distributed scheduling in the asynchronous case.

As the *low-level complement* q^c of a port q (either in- or output port) we denote the port with which it connects according to Figure 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. The *high-level complement* q^C of a port q denotes the connecting port without the buffer, i.e., $\mathsf{q}!^C = \mathsf{q}?$ and vice versa. For a set or a sequence P of ports, let $\mathsf{in}(P)$ and $\mathsf{out}(P)$ denote the subset or subsequence of P consisting of the input ports or the output ports of P, respectively.

After introducing ports, we now focus on the definition of *machines*. A machine has a *sequence of ports*, containing both input ports and output ports, and a set of *states*, comprising sets of *initial* and *final states*. 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. Furthermore, each machine has a bound on the length of the considered inputs which allows time bounds on the computation time independent of the environment. The parts of an input that are beyond the length bound are ignored, i.e., incoming strings are only processed up to a predefined length. In particular, this is used to ensure polynomial runtime of individual machines.

Definition 1. *(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 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 $name_{\mathsf{M}}$. For a set $\hat{M}$ of machines, let $\mathsf{ports}(\hat{M})$ denote the set of ports of all machines $\mathsf{M} \in \hat{M}$. 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, usually a security parameter k given in unary representation; in particular, polynomial-time is meant in this sense. We only briefly state here, that these machines have a natural realization as a probabilistic interactive Turing machine as introduced in [16]. 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.

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 $P := (\mathsf{p}_1?, \dots, \mathsf{p}_n?) \subseteq \mathsf{in}(Ports_{\mathsf{M}})$ be a subsequence of the input ports of M and $(v_i)_{i\in\{1,\dots,n\}} \in (\Sigma^*)^n$. Then $\mathcal{I}_{\mathsf{p}_1?=v_1,\dots,\mathsf{p}_n?=v_n}$ denotes the input with $\mathsf{p}_i? = v_i$ for all i and $\mathsf{p}'? = \epsilon$ for all $\mathsf{p}'? \in \mathsf{in}(Ports_{\mathsf{M}}) \setminus P$. In the special case $\mathsf{p}_i? = \epsilon$ for all i, i.e., in case of an all-empty input, we write $\mathcal{I}_\epsilon$. Outputs are defined similarly.

A *collection* $\mathcal{C}$ of machines is a finite set of machines with pairwise different machine names and disjoint sets of ports. In the asynchronous model, the *completion* $[\mathcal{C}]$ of a collection $\mathcal{C}$ is the union of all machines of $\mathcal{C}$ and the buffers needed for every channel. A port of a collection is called *free* if its connecting port is not in the collection. These port will be connected to the users and the adversary. The free ports of a collection $\mathcal{C}$ are denoted as $\mathsf{free}(\mathcal{C})$. In the asynchronous model, a collection $\mathcal{C}$ is called *closed* if its completion $[\mathcal{C}]$ has no free ports except a special master clock-in port $\mathsf{clk}^{\triangleleft}?$, i.e., $\mathsf{free}([\mathcal{C}]) = \{\mathsf{clk}^{\triangleleft}?\}$. When we define the interaction of several machines, this port will be used to resolve situations where the interaction cannot proceed. In the synchronous case, we demand $\mathsf{free}(\mathcal{C}) = \emptyset$.

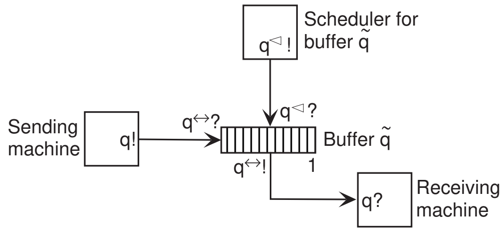

Fig. 1. Ports and buffers.

For security purposes, special collections are needed, because an adversary may have taken over parts of the initially intended system, e.g., different situations have to be captured depending on which and how many users are considered as being malicious. Therefore, a system consists of several possible remaining structures.

Definition 2. *(Structures and Systems) A* structure *is a pair* $struc = (\hat{M}, S)$ *where* $\hat{M}$ *is a collection of non-buffer 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*, i.e., those ports that the honest user is forbidden to have. A* system *Sys is a set of structures. It is polynomial-time iff all machines in all its collections* $\hat{M}$ *are polynomial-time.* ◇

The separation of the free ports into specified ports and others is an important feature of the upcoming security definitions. The specified ports are those where a certain service is guaranteed.

Note that this definition is valid for both the synchronous and the asynchronous case. In particular, buffers do not have to be explicitly included in the specification of a system, e.g., in the specification of a cryptographic protocol that one wants to analyze. The different timing assumption stem from the different definitions of runs which we will introduce in the following.

A structure can be completed to a *configuration* by adding machines H and A, modeling the joint 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. In the asynchronous case, buffers are additionally added to close the collection.

Definition 3. *(Configurations) 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,* $\hat{M} \cup \{\mathsf{H}, \mathsf{A}\}$ *is a closed collection, and* $\mathsf{ports}(\mathsf{H}) \cap \mathsf{forb}(\hat{M}, S) = \emptyset$*. The* *set of configurations is written* $\mathsf{Conf}(Sys)$*. 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. The initial states of all machines in a configuration are a common security parameter* k *in unary representation.* $\diamond$

2.2 Capturing Asynchronous Runs

For a configuration, both models define a probability space of runs (sometimes called *traces* or *executions*). In the asynchronous model, scheduling of machines is done sequentially, so we have exactly one active machine M at any time. If this machine has clock-out ports, it is allowed to select the next message to be scheduled as explained at the beginning of Section 2.1. If this message exists, it is delivered by the buffer and the unique receiving machine is the next active machine. If M tries to schedule multiple messages, only one is taken, and if it schedules none or the message does not exist, the special master scheduler is scheduled. This is formally captured as follows.

Definition 4. *(Asynchronous Runs and Views) For a given configuration* $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A})$ *with master scheduler* $\mathsf{X} \in \hat{M} \cup \{\mathsf{A}\}$*, set* $\hat{C} := [\hat{M} \cup \{\mathsf{H}, \mathsf{A}\}]$*. 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 input port values* I*. Then assign* ϵ *to all input ports of* $\mathsf{M_{CS}}$*.*
2. Termination: *If* X *is in a final state, the run stops.*
3. Buffer messages: *For each simple output port* $\mathsf{q}!$ *of* $\mathsf{M_{CS}}$*, in their given order, switch buffer* $\widetilde{\mathsf{q}}$ *with input* $\mathsf{q}^{\leftrightarrow}? := \mathsf{q}!$*, cf. Figure 1. Then assign* ϵ *to all these ports* $\mathsf{q}!$ *and* $\mathsf{q}^{\leftrightarrow}?$*.*
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* $\widetilde{\mathsf{q}}$ *with input* $\mathsf{q}^{\triangleleft}? := \mathsf{q}^{\triangleleft}!$ *(cf. Figure 1), set* $\mathsf{q}? := \mathsf{q}^{\leftrightarrow}!$ *and then assign* ϵ *to all ports of* $\widetilde{\mathsf{q}}$ *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) with name $name_{\mathsf{M}}$ *is switched from* (s, I) *to* (s', O)*, we add a* step $(name_{\mathsf{M}}, s, I', s', O)$ *with* $I' := I\lceil_{I_{\mathsf{M}}(s)}$ *to the run* r*, except if* s *is final or* $I' = (\epsilon, \ldots, \epsilon)$*. This gives a random variable for each value of the security parameter denoted as* $run_{conf,k}$*, hence we obtain a family of random variables*

$$run_{conf} = (run_{conf,k})_{k \in \mathbb{N}}.$$

The view *of a subset* $M \subset \hat{C}$ *in a run* r *is the restriction of* r *to* 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* $\mathsf{M} \in M$*. This gives a family of random variables*

$$view_{conf}(M) = (view_{conf,k}(M))_{k \in \mathbb{N}}.$$

For a singleton $M = \{\mathsf{H}\}$ *we write* $view_{conf}(\mathsf{H})$ *instead of* $view_{conf}(\{\mathsf{H}\})$*.* $\diamond$

This rather informal definition of runs can naturally be formalized using transition probabilities, which induce probability spaces over the finite sequences of steps similar to Markov Chains. The extension to infinite sequences can then be achieved using well-established results of measure theory and probability theory, cf. Section 5 of [31]. It is further easy to show that views of polynomial-time machines are of polynomial size.

2.3 Capturing Synchronous Runs

In the synchronous model, ports, machines, collections, structures, and systems are defined similar to the asynchronous model. The only exception is that there are no clock ports and no buffers, which have only been included to model asynchronous timing, i.e., corresponding ports $\mathsf{p}?$ and $\mathsf{p}!$ are directly connected. The main difference is the definition of runs. Instead of our asynchronous run algorithm (cf. Definition 4), 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, called *clocking scheme*, 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.

Definition 5. *(Clocking Scheme) A clocking scheme κ for a configuration $(\hat{M}, S, \mathsf{H}, \mathsf{A})$ and $n \in \mathbb{N}$ is a mapping from the set $\{1, \ldots, n\}$ to the powerset of $\hat{M} \cup \{\mathsf{H}, \mathsf{A}\}$, i.e., it assigns each number a subset of the machines.* $\diamond$

Definition 6. *(Synchronous Runs and Views) Given a configuration $conf = (\hat{M}, S, \mathsf{H}, \mathsf{A})$ along with a clocking scheme κ for $n \in \mathbb{N}$, 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 switched. If several inputs arrive until that time, they are concatenated. This gives a family of random variables*

$$run_{conf} = (run_{conf,k})_{k \in \mathbb{N}}.$$

More precisely, each run *is a function mapping each triple $(\mathsf{M}, i, j) \in \hat{M} \cup \{\mathsf{H}, \mathsf{A}\} \times \mathbb{N} \times \{1, \ldots, n\}$ to a quadruple (s, I', s', O) of the old state, inputs (with $I' := I\lceil_{l_{\mathsf{M}}(s)}$ again), new state, and outputs of machine M in subround $[i.j]$, with $I' \equiv \epsilon$, $O \equiv \epsilon$, and $s = s'$ if M is not switched in this subround. The* view *of a subset $M \subset \hat{M} \cup \{\mathsf{H}, \mathsf{A}\}$ in a run r is the restriction of r to $M \times \mathbb{N} \times \{1, \ldots, n\}$. This gives a family of random variables*

$$view_{conf}(M) = (view_{conf,k}(M))_{k \in \mathbb{N}}.$$

$\diamond$

Again, the view of a polynomial-time machine can easily be shown to be of polynomial size.

Remark 1. 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 proofs.

Instead of arbitrary clocking schemes as in the above definition of runs, the authors of [35] focus on only one special clocking scheme κ, 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 [35], 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 via one duplex channel. This is indeed no restriction to generality in the synchronous model, because outputs at several ports to the same machine can simply be concatenated using a separation symbol and decomposed again, respectively. In the following, we give these two channels fixed names $\mathsf{p}_{\mathsf{A\_H}}$ and $\mathsf{p}_{\mathsf{H\_A}}$, i.e., $\mathsf{p}_{\mathsf{A\_H}}!$ sends messages from A to H and vice versa.

2.4 Simulatability

The definition of one system securely implementing another one is based on the common concept of *simulatability*. Simulatability essentially means that whatever might happen to an honest user in a real system Sys_{real} can also happen in an ideal (abstract) system Sys_{id}: For every structure $struc_1 \in Sys_{\mathsf{real}}$, every user H, and every adversary A_1, there exists an adversary A_2 on a corresponding ideal structure $struc_2$ such that the view of H is indistinguishable in the two configurations. Indistinguishability ("$\approx$") is a well-defined cryptographic notion from [43]. We only give the definition of computational indistinguishability; a more comprehensive definition is given in the Appendix.

Definition 7. *(Computational 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* computationally indistinguishable *("$\approx$") if for every algorithm* Dis *(the distinguisher) that is probabilistic polynomial-time in its first input,*

$$|P(\mathsf{Dis}(1^k, \mathsf{var}_k) = 1) - P(\mathsf{Dis}(1^k, \mathsf{var}'_k) = 1)| \in \mathit{NEGL}.^1$$

Intuitively, given the security parameter and an element chosen according to either var_k *or* var'_k, Dis *tries to guess which distribution the element came from.* $\diamond$

Corresponding structures in the simulatability definition are designated by a function f from Sys_{real} to the powerset of Sys_{id}. The function f is called *valid* if it maps structures with the same set of specified ports. We only give the definition of simulatability based on computational indistinguishability, which captures the most common case when applying simulatability to cryptographic protocols. A more comprehensive definition based on the remaining notions of indistinguishability is again postponed to the Appendix; our results hold as well for this more general definition.

Definition 8. *(Simulatability) Let systems* Sys_1 *and* Sys_2 *with a valid mapping* f *be given. We say* $Sys_1 \geq^f Sys_2$ (at least as secure as) *if for every polynomial-time configuration* $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1)$*, there exists a polynomial-time 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* $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$. $\diamond$

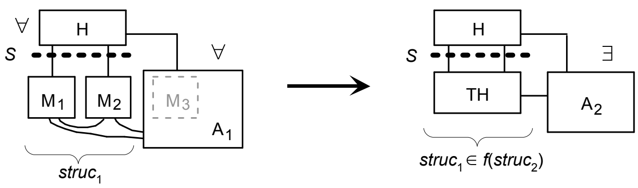

Fig. 2. Overview of the simulatability definition.

This is shown in Figure 2. In the following, we augment $\geq$ with a subscript $_{\mathsf{sync}}$ or $_{\mathsf{async}}$ to distinguish the definition of the synchronous and asynchronous case. In a typical ideal system, each structure contains only one machine TH called trusted host, which serves as an ideal functionality of the real system. The machine TH is usually deterministic and maintains a very simple transition function, hence validation based on this ideal functionality is in scope of current verification techniques.

3 Idea and Definition of the Embedding

The informal idea of the embedding φ_{Sys} is to add an explicit master scheduler that should simulate the synchronous run induced by the given clocking scheme. However, due to the general distributed scheduling (cf.

[1] The class NEGL denotes the set of all negligible functions, i.e., $g\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \in \mathit{NEGL}$ if for all positive polynomials Q, $\exists k_0 \forall k \geq k_0\colon\ g(k) \leq 1/Q(k)$.

Definition 4), leaving the actual machines unmodified leads to non-simulatable situations, as these machines can clock themselves without ever giving control to this explicit master scheduler.

Hence, we first define a mapping φ_{M} that surrounds single synchronous machines (i.e., machines that are designed for a synchronous environment) with an "asynchronous coat". More precisely, if a synchronous machine makes a transition, it obtains all inputs at once that arrived since its last scheduling, whereas in asynchronous scenarios, these inputs come one by one and have to be processed in several transitions. Thus, the surrounding asynchronous machine stores all inputs internally, until it is asked to perform the transition of its synchronous submachine. It then schedules this submachine with the collected inputs and forwards its outputs. As these asynchronous machines do not produce any clock outputs, the master scheduler can try to simulate the synchronous time by a suitable scheduling strategy.

Definition 9 (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 $Ports_{\mathsf{M}_{\mathsf{async}}} := Ports_{\mathsf{M}_{\mathsf{sync}}} \circ (\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}?)$, where $\circ$ denotes concatenation of sequences.
- Internally, $\mathsf{M}_{\mathsf{async}}$ maintains arrays $(input\_store_{\mathsf{M}_{\mathsf{sync}},\mathsf{p}?})_{\mathsf{p}? \in \mathsf{in}(Ports_{\mathsf{M}_{\mathsf{sync}}})}$ over Σ^* initialized with ϵ everywhere, which are used for storing incoming messages at each port of $\mathsf{M}_{\mathsf{sync}}$.
- $\mathsf{M}_{\mathsf{async}}$ has the machine $\mathsf{M}_{\mathsf{sync}}$ as a blackbox submachine, i.e., it has its transition function $\delta_{\mathsf{M}_{\mathsf{sync}}}$.
- Internally, $\mathsf{M}_{\mathsf{async}}$ maintains exactly the states of $\mathsf{M}_{\mathsf{sync}}$. Moreover, the initial and final states of both machines are equal.
- On input i at $\mathsf{p}? \neq \mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}?$: It concatenates i to the element of $input\_store_{\mathsf{M}_{\mathsf{sync}},\mathsf{p}?}$, i.e., it stores all inputs until the machine $\mathsf{M}_{\mathsf{sync}}$ is eventually switched.
- On 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 $(s', \mathcal{O})$. $\mathsf{M}_{\mathsf{async}}$ now assigns ϵ to $input\_store_{\mathsf{M}_{\mathsf{sync}},\mathsf{p}?}$ for all $\mathsf{p}? \in \mathsf{in}(Ports_{\mathsf{M}_{\mathsf{sync}}})$, switches to the state s' and outputs the tuple $\mathcal{O}$. This case corresponds to the scheduling of the synchronous machine; the port $\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}?$ will be connected to the explicit master scheduler.

Obviously, $\mathsf{M}_{\mathsf{async}}$ is polynomial-time by construction iff $\mathsf{M}_{\mathsf{sync}}$ is polynomial-time, since both machines always stay in the same state after a transition and their final states are equal. Moreover, we define the function φ_{M} on a set $\hat{M}$ of synchronous machines by $\varphi_{\mathsf{M}}(\hat{M}) := \bigcup_{\mathsf{M}_{\mathsf{sync}} \in \hat{M}} \varphi_{\mathsf{M}}(\mathsf{M}_{\mathsf{sync}})$. $\diamond$

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

Definition 10 (Mapping φ_{Sys}). Let an arbitrary synchronous system $Sys_{\mathsf{sync}} = \{(\hat{M}_{\mathsf{sync}}, S_{\mathsf{sync}}) \mid \mathsf{sync} \in I\}$ for a finite index set I and a clocking scheme κ be given. We then define

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

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

- $\{\mathsf{clk}^{\triangleleft}?\}$: The master clock-in port.
- $\{\mathsf{p}^{\triangleleft}! \mid \mathsf{p}! \in Ports_{\hat{M}_{\mathsf{sync}}}\}$: Ports for clocking all output ports of the given structure.
- $\{\mathsf{p}^{\triangleleft}! \mid \mathsf{p}? \in \mathsf{free}(\hat{M}_{\mathsf{sync}})\}$: 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.[2]
- $\{\mathsf{p}_{\mathsf{M}}!, \mathsf{p}_{\mathsf{M}}{}^{\triangleleft}! \mid \mathsf{M} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}, \mathsf{A}\})\}$: Ports for clocking, i.e., giving control to, each machine.

[2] 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 2.3).

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 behavior 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 $\mathsf{p_M}!$, 1 at $\mathsf{p_M}^{\triangleleft}!$. The order of the switched machines can be chosen arbitrary.
2. **Schedule Outgoing Buffers:** For all $\mathsf{M} \in \kappa(local\_rnd)$ output 1 at every port $\mathsf{p}^{\triangleleft}!$ with $\mathsf{p}! \in Ports_{\mathsf{M}}$. Here, the order of the switched machines can only be chosen arbitrary with the restriction that output ports of the adversary are scheduled first if $\mathsf{A} \in \kappa(local\_rnd)$.[3]
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).

$\diamond$

To put it all into a nutshell, the specific 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). Finally, it switches to the next subround (Step 3) and continues with the first step again.

Moreover, we define a mapping φ_{conf} on synchronous configurations of a system Sys, i.e., configurations which consist of synchronous machines only, by

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

with $\mathsf{X}_{\mathsf{sync},\kappa}$ given as in φ_{Sys} for the particular structure. We will in the following simply write φ instead of $\varphi_{Sys}, \varphi_{\mathsf{M}}$, and φ_{conf} if its meaning is clear from the context.

4 Preliminary Work for the Embedding Theorems

We now have to prove that the function φ has the desired properties with respect to simulatability, i.e.,

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

This captures the content of our first embedding theorem. Unfortunately, the converse direction does not hold, but our second embedding theorem will state a weaker version that is still sufficient for our purpose.

4.1 Proof Overview

Before we turn our attention to the auxiliary lemmas for the embedding theorems we exemplarily present an informal description of the proof of the first embedding theorem. The proof consists of four steps. A graphical illustration is given in Figure 3.

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 define a mapping ϕ on the runs of the asynchronous system yielding runs of the synchronous system. Intuitively, ϕ "compresses" an asynchronous run to its synchronous counterpart, which consists of much less steps. We then show in Theorem 1 that $view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$.

[3] Without this restriction, the behavior of the adversary at its switching time could depend on outputs of machines scheduled in the same subround, which would lead to non-simulatable situations.

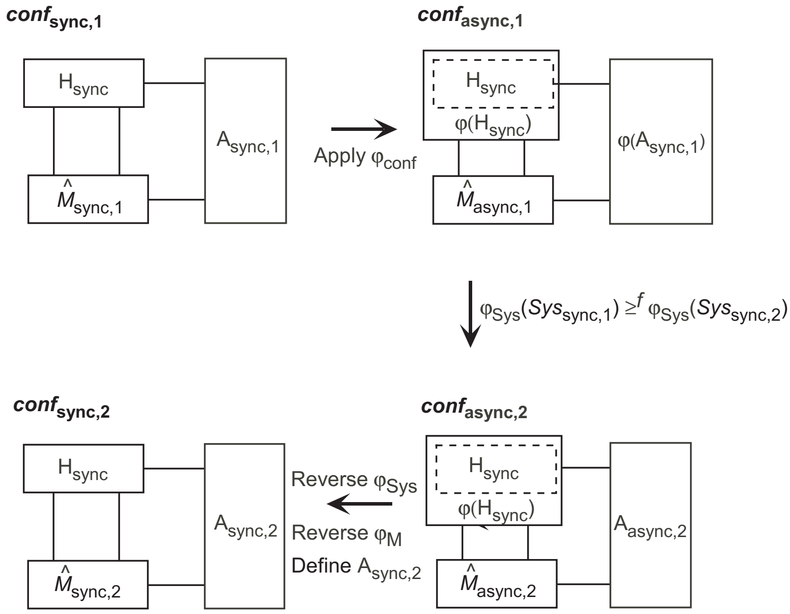

Fig. 3. Synchronous Simulatability derived by Asynchronous Simulatability.

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 $conf_{\mathsf{async},2} \in \mathsf{Conf}(\varphi_{Sys}(Sys_{\mathsf{sync},2}))$, written $view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})) \approx view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))$. We then show that $\phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) \approx \phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$.
3. We finally reverse the function φ by removing the coating of the user and the machines of the structure. Since we do not know anything about the newly derived adversary $\mathsf{A}_{\mathsf{async},2}$, i.e., it is not forced to fit the structure imposed by the mapping φ, 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 2 that $\phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) = view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})$.
4. Altogether, transitivity of the relation $\approx$ implies $view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}})) \approx view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})$.

We first take a look at the runs in a synchronous system Sys_{sync} and in its asynchronous counterpart $Sys_{\mathsf{async}} := \varphi(Sys_{\mathsf{sync}})$. 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.

4.2 Compressing asynchronous runs to synchronous counterparts

In the following, 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).[4]

First of all, note that runs of $conf_{\mathsf{async}}$ always have a prescribed structure induced by the behavior 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.

[4] Note that we investigate the more general case here that A' can be chosen arbitrarily instead of being the embedded adversary $\varphi_{\mathsf{M}}(\mathsf{A})$. This generality will be helpful during the upcoming proofs.

1. The first block consists of the steps induced by clocking the machines $\varphi(\mathsf{M}_{\mathsf{sync}})$ with $\mathsf{M}_{\mathsf{sync}} \in \kappa(j)$ and A' if $\mathsf{A}_{\mathsf{sync}} \in \kappa(j)$, 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 $(\mathsf{X}_{\mathsf{sync},\kappa}, s_1, \mathcal{I}_{\mathsf{clk}^{\triangleleft}?=1}, s_1', \mathcal{O}_{\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}!=(i,j),\mathsf{p}_{\mathsf{M}_{\mathsf{sync}}}{}^{\triangleleft}!=1})$ for two arbitrary states s_1, s_1' of $\mathsf{X}_{\mathsf{sync},\kappa}$, i.e., the master scheduler schedules the machine $\varphi(\mathsf{M}_{\mathsf{sync}})$ respectively A'.
   - After that, we have the transition of the scheduled buffer.
   - We now have to distinguish the following two cases:
     - If $\mathsf{M}_{\mathsf{sync}} \neq \mathsf{A}_{\mathsf{sync}}$, 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 $\mathsf{M}_{\mathsf{sync}} = \mathsf{A}_{\mathsf{sync}}$, we have a step $(\mathsf{A}', s, \mathcal{I}_{\mathsf{p}_{\mathsf{A}_{\mathsf{sync}}}?=(i,j)}, s', \mathcal{O})$. If $\mathcal{O} \neq \mathcal{O}_\epsilon$ we have steps for the receiving buffers. If there are nonempty outputs at ports p! and $\mathsf{p}^{\triangleleft}!$ (which has to be a self-loop because there are no free clock-in 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 divergence, 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 $(\mathsf{X}_{\mathsf{sync},\kappa}, s_1, \mathcal{I}_{\mathsf{clk}^{\triangleleft}?=1}, s_1', \mathcal{O}_{\mathsf{p}^{\triangleleft}!=1})$ 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 two cases:
     - If $\mathsf{M}' \neq \mathsf{A}'$, 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 p!.
     - If $\mathsf{M}' = \mathsf{A}'$, we obtain a step $(\mathsf{A}', s, \mathcal{I}', s', \mathcal{O})$, where $\mathcal{I}'$ consists of the output of $\varphi(\mathsf{M}_{\mathsf{sync}})$ respectively A' at p!. If $\mathcal{O} \neq \mathcal{O}_\epsilon$ we have steps for the receiving buffers. If $\mathcal{O}$ has a clocked self-loop, we proceed identical to the first block.
   - 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., its 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 $\mathsf{M}_{\mathsf{sync}}$ that switches in the j-th subround of the particular global round.

Definition 11. *(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 $\mathsf{p}? \in \mathsf{in}(Ports_{\mathsf{M}})$. It goes from block to block modifying them as follows.*

1. *Every step of a buffer is deleted from the run.*
2. *The two remaining steps of the first block are modified as follows. If the scheduled machine is $\varphi(\mathsf{M}_{\mathsf{sync}}) \neq \mathsf{A}'$, then the block is 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 scheduled, 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' when it is 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 divergence, the algorithm for defining the mapping ϕ diverges, too.).*

3. *The algorithm starts searching through 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* $inputs_{\mathsf{M}',\mathsf{p}?}$.
4. *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 block except for the inputs of each machine in the synchronous case. At this point, it also becomes clear why we defined the master scheduler to schedule each machine specifically with a tuple (i, j) indicating the current global and local round, since this information would otherwise not be contained in the asynchronous run.

To overcome the absence of the gathered inputs in the run, the algorithms has to collect all "partial" inputs itself in its third step, and it can use this information to calculate the outputs of each machine (although for this, it could as well use the information contained in the run). 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 2. Furthermore, the following lemma establishes a computational bound on the mapping ϕ in polynomial-time configurations:

Lemma 1. *If* $conf_{\mathsf{async}}$ *is a polynomial-time configuration that fits the form required by Definition 11, then* ϕ *applied to the view of the honest user and the adversary is computable in polynomial-time.* □

4.3 Auxiliary Theorems

The following theorem captures the first step of our proofsketch of Section 4.1.

Theorem 1. *Let a synchronous system* Sys_{sync}*, a clocking scheme* κ*, and a 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, and set* $conf_{\mathsf{async}} := \varphi(conf_{\mathsf{sync}})$*. Then*

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

for every $\mathsf{M}_{\mathsf{sync}} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}}\})$. $conf_{\mathsf{async}}$ *is polynomial-time iff* $conf_{\mathsf{sync}}$ *is polynomial-time.* □

After performing this first step of the proof, 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 (again postponing its proof to the Appendix).

Theorem 2. *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. 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. Then there exists an adversary* $\mathsf{A}_{\mathsf{sync}}$ *using* $\mathsf{A}_{\mathsf{async}}$ *as a blackbox such that for* $conf_{\mathsf{sync}} := (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync}})$*, it holds*

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

for every $\mathsf{M}_{\mathsf{sync}} \in (\hat{M}_{\mathsf{sync}} \cup \{\mathsf{H}_{\mathsf{sync}}\})$. $conf_{\mathsf{async}}$ *is polynomial-time iff* $conf_{\mathsf{sync}}$ *is polynomial-time.* □

Note, that the standard clocking scheme $(\hat{M} \cup \{\mathsf{H}\}, \{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{A}\})$ fulfills the postulated requirement.

5 The Embedding Theorems

This section contains our two main theorems. We start with a lemma capturing some simple properties of indistinguishable random variables. The lemma is well-known and easily proved.

Lemma 2 (Indistinguishability). *Indistinguishability of two families of random variables implies indistinguishability of any function ϕ of them. For the polynomial case, the function ϕ has to be polynomial-time computable. Moreover, identically distributed variables are indistinguishable and indistinguishability is an equivalence relation.* □

Theorem 3. *(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 $(\hat{M}_2, S_2) \in f'(\hat{M}_1, S_1) \Leftrightarrow \varphi(\hat{M}_2, S_2) \in f(\varphi(\hat{M}_1, S_1))$. □

Using the result of the previous theorems, the proof will be rather simple, cf. the illustration in Figure 3.

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 1, applying the mapping ϕ to the runs of $conf_{\mathsf{async},1}$ yields
$$view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$
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. Thus, the precondition $\varphi(Sys_{\mathsf{sync},1}) \geq^f_{\mathsf{async}} \varphi(Sys_{\mathsf{sync},2})$ 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
$$view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})) \approx view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))$$
and $\varphi(\hat{M}_{\mathsf{sync},2}, S) \in f(\varphi(\hat{M}_{\mathsf{sync},1}, S))$. Moreover, in the computational case, $conf_{\mathsf{async},2}$ is polynomial-time, so the mapping ϕ is polynomial-time computable. Using Lemma 2, 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 now apply Theorem 2 to the configuration $conf_{\mathsf{async},2}$, which yields a configuration $conf_{\mathsf{sync},2} = (\hat{M}_{\mathsf{sync}}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{sync},2}) \in \mathsf{Conf}(Sys_{\mathsf{sync},2})$ with
$$\phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) = view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}}).$$
According to Theorem 2, $conf_{\mathsf{sync},2}$ is a polynomial-time configuration iff $conf_{\mathsf{async},2}$ is polynomial.
4. Putting it all together, we have
   - $view_{conf_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) = \phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$
   - $\phi(view_{conf_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) \approx \phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$ and
   - $\phi(view_{conf_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) = view_{conf_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})$.

Using Lemma 2, 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 $conf_{\mathsf{sync},1}$. Moreover, we have $\varphi(\hat{M}_{\mathsf{sync},2}, S) \in f(\varphi(\hat{M}_{\mathsf{sync},1}, S))$, i.e., $(\hat{M}_{\mathsf{sync},2}, S) \in f'(\hat{M}_{\mathsf{sync},1}, S)$ which yields the desired result $Sys_{\mathsf{sync},1} \geq^{f'}_{\mathsf{sync}} Sys_{\mathsf{sync},2}$.

■

Note that the theorem is applicable to the standard clocking scheme. So far, we have shown that asynchronous simulatability among these asynchronous representations implies synchronous simulatability, i.e.,

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

We already briefly stated in the previous section that the converse implication does not hold in general. We had to show that for each 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 $Sys_{\mathsf{sync},1} \geq_{\mathsf{sync}} Sys_{\mathsf{sync},2}$.

However, both the honest user and the adversary may have clock-out ports and they can alternately schedule each other (and also the system erratically), which we cannot capture by a fixed 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 version of simulatability for the restricted class of users by $\geq_{\mathsf{async},\mathsf{H}}$ in the sequel. Looking at the proof of the first embedding theorem, it is immediately clear that the theorem also holds for the weaker precondition $\varphi_{Sys}(Sys_{\mathsf{sync},1}) \geq_{\mathsf{async},\mathsf{H}} \varphi_{Sys}(Sys_{\mathsf{sync},2})$, since we only need to derive an indistinguishable configuration for users of the special form $\varphi(\mathsf{H}_{\mathsf{sync}})$, and the user remains unchanged at simulatability. We can now capture the content of the second embedding theorem as

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

Theorem 4. *(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*

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

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

6 Deriving Synchronous Theorems from Asynchronous Ones

Recall that our long-term goal is to avoid proving each and every theorem and lemma for both models. We now briefly show how our two embedding theorems can be used for circumventing this problem. One of the most important theorems of both models is transitivity of the relation $\geq$.

Lemma 3 (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 as $f_3(\hat{M}_1, S)$ being the union of the sets $f_2(\hat{M}_2, S)$ with $(\hat{M}_2, S) \in f_1(\hat{M}_1, S)$.* □

This has been proven in [35] for the synchronous and in [37] for the asynchronous model. We now exemplarily show how to derive the synchronous version from the asynchronous one using our previous results.

Lemma 4. *(Asynchronous Version of Transitivity implies Synchronous Version) Assume that the asynchronous version of the transitivity lemma (Lemma 3) has already been proven, then the synchronous version holds as well.* □

Proof. We omit the superscripts f_i for the sake of readability. Let 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

$$\varphi(Sys_1) \geq_{\mathsf{async,H}} \varphi(Sys_2) \quad \text{and} \quad \varphi(Sys_2) \geq_{\mathsf{async,H}} \varphi(Sys_3).$$

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 its subsequent remarks (stating that the theorem holds as well for the restricted version $\geq_{\mathsf{async,H}}$ of simulatability) yielding $Sys_1 \geq_{\mathsf{sync}} Sys_3$. ■

This proof technique is applicable to almost all theorems that rely on simulatability. As the most important example, we name the preservation theorem [36, 4], which states that integrity properties expressed in linear-time logic are preserved under simulatability. The proof of this theorem is difficult and comprises several pages for both models. Using our work, the synchronous proof could as well be omitted.

However, this proof techniques is unfortunately not immediately applicable to carry over lemmas dealing with composition of systems, since it is not immediately clear what the result of composing two systems with different master schedulers is. This problem can probably be circumvented as follows. First, both master schedulers are combined to an overall scheduler X for the whole system. Secondly, an intermediate system can be defined, where this combined master scheduler is split into two separate machines X_1 and X_2 such that X_1 stays the true master scheduler with the unique master clock-in port $\mathsf{clk}^{\triangleleft}?$, and X_2 is considered as a "slave" master scheduler, i.e., a usual machine that is explicitly given control by X_2 to handle the scheduling demands of "its" system. Finally, our embedding theorems are applicable in this intermediate system, and the resulting schedulers can be composed again to an overall master scheduler. However, formally establishing this result requires additional research.

Acknowledgments

This work benefited from fruitful discussions with *Birgit Pfitzmann* and *Michael Waidner*.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
2. M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 82–94, 2001.
3. M. Abadi and P. Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. In *Proc. 1st IFIP International Conference on Theoretical Computer Science*, volume 1872 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.
4. M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *Proc. 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 2607 of *Lecture Notes in Computer Science*, pages 675–686. Springer, 2003.
5. M. Backes, C. Jacobi, and B. Pfitzmann. Deriving cryptographically sound implementations using composition and formally verified bisimulation. In *Proc. 11th Symposium on Formal Methods Europe (FME 2002)*, volume 2391 of *Lecture Notes in Computer Science*, pages 310–329. Springer, 2002.

6. M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive 2003/015, Jan. 2003. `http://eprint.iacr.org/`.
7. D. Beaver. Secure multiparty protocols and zero knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4(2):75–122, 1991.
8. M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 419–428, 1998.
9. M. Burrows, M. Abadi, and R. Needham. A logic for authentication. Technical Report 39, SRC DIGITAL, 1990.
10. R. Canetti. Studies in secure multiparty computation and applications. Department of Computer Science and Applied Mathematics, The Weizmann Institute of Science, June 1995, revised March 1996, 1995.
11. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 3(1):143–202, 2000.
12. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.
13. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
14. C. Dwork, M. Naor, and A. Sahai. Concurrent zero-knowledge. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 409–418, 1998.
15. S. Goldwasser and L. Levin. Fair computation of general functions in presence of immoral majority. In *Advances in Cryptology: CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 1990.
16. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–207, 1989.
17. J. D. Guttman, F. J. Thayer Fabrega, and L. Zuck. The faithfulness of abstract protocol analysis: Message authentication. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 186–195, 2001.
18. M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.
19. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science, Prentice Hall, Hemel Hempstead, 1985.
20. R. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, 1989.
21. R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.
22. P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. 10th European Symposium on Programming (ESOP)*, pages 77–91, 2001.
23. P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proc. 5th ACM Conference on Computer and Communications Security*, pages 112–121, 1998.
24. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.
25. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, 1996.
26. C. Meadows. Using narrowing in the analysis of key management protocols. In *Proc. 10th IEEE Symposium on Security & Privacy*, pages 138–147, 1989.
27. C. Meadows. Formal verification of cryptographic protocols: A survey. In *Proc. ASIACRYPT '94*, volume 917 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1994.
28. S. Micali and P. Rogaway. Secure computation. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 392–404. Springer, 1991.
29. J. K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proc. 5th IEEE Symposium on Security & Privacy*, pages 134–141, 1984.
30. B. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.
31. J. Neveu. *Mathematical Foundations of the Calculus of Probability*. Holden-Day, 1965.
32. S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *Proc. 11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. springer, 1992.
33. L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.
34. B. Pfitzmann, M. Schunter, and M. Waidner. Cryptographic security of reactive systems. Presented at the DERA/RHUL Workshop on Secure Architectures and Information Flow, 1999, Electronic Notes in Theoretical Computer Science (ENTCS), March 2000. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/menu.htm`.
35. B. Pfitzmann, M. Schunter, and M. Waidner. Secure reactive systems. Research Report RZ 3206, IBM Research, 2000.
36. B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *Proc. 7th ACM Conference on Computer and Communications Security*, pages 245–254, 2000.

37. B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. 22nd IEEE Symposium on Security & Privacy*, pages 184–200, 2001.
38. R. Segala and N. Lynch. Probabilistic simulation for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.
39. F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. 19th IEEE Symposium on Security & Privacy*, pages 160–171, 1998.
40. D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. 27th Symposium on Principles of Programming Languages (POPL)*, pages 268–276, 2000.
41. S.-H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic I/O automata. *Theoretical Computer Science*, 176(1–2):1–38, 1997.
42. A. C. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 160–164, 1982.
43. A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 1982.

A Postponed Definitions

The following definition for indistinguishability of random variables is essentially from [43].

Definition 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 suitable 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 affine addition, and with a function* g *also contain every function* $g' \leq g$*.*

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

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

Intuitively, given the security parameter and an element chosen according to either var_k *or* var'_k*,* Dis *tries to guess which distribution the element came from. The class* $NEGL$ *denotes the set of all negligible functions, i.e.,* $g\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \in NEGL$ *if for all positive polynomials* Q*,* $\exists k_0 \forall k \geq k_0\colon\; g(k) \leq 1/Q(k)$*.*

We write $\approx$ *if we want to treat all three cases simultaneously.* ◇

For reasons of completeness, we now present the extended definition of simulatability, based on the three different kinds of indistinguishability. Definition 8 was simplified in the sense that only computational indistinguishability of views was covered, which represents the most common case when applying simulatability to cryptographic protocols.

Definition 13. *(Simulatability) Let systems* Sys_1 *and* Sys_2 *with a valid mapping* f *be given.*

a) We say $Sys_1 \geq_{\mathsf{sec}}^{f,\mathsf{perf}} Sys_2$ (perfectly at least as secure as*) if for every configuration* $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_1)$*, 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* $\mathsf{Conf}_{\mathsf{poly}}(Sys_1)$ *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.* ◇

B Postponed Proofs

Proof. (Lemma 1) 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 to 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.[5] This implies that ϕ is computable in polynomial-time when applied to the view of the honest user and the adversary if it is used in a polynomial-time configuration. ■

Proof. (Theorem 1) Note that the view of $\varphi(\mathsf{M_{sync}})$ 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 $\mathsf{H_{sync}}$ in $conf_{\mathsf{sync}}$ are equal. It is quite easy to see that the arrays $input\_store_{\mathsf{M_{sync}}}$ and $inputs_{\mathsf{M_{sync}}}$ are always equal if the machine $\mathsf{M_{sync}}$ is switched. This can easily be proven by induction over the number of (sub-)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 $\mathsf{M_{sync}}$. However, these inputs only depend on *prior* outputs of other machines M. Moreover, these outputs have to be 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 $inputs_{\mathsf{M_{sync}}}$ and $input\_store_{\mathsf{M_{sync}}}$ must be equal at replacing the block by construction of the algorithm, so $\delta_{\mathsf{M_{sync}}}(s, inputs_{\mathsf{M_{sync}}}) = \delta_{\mathsf{M_{sync}}}(s, input\_store_{\mathsf{M_{sync}}})$ 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 did 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

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

[5] 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.

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. ∎

Proof. (Theorem 2) 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 behavior is defined as follows. If $\mathsf{A}_{\mathsf{sync}}$ is clocked in the synchronous system, it gets an input tuple $\mathcal{I} = (\mathcal{I}_{\mathsf{p?}})_{\mathsf{p?} \in \mathsf{in}(\mathsf{ports}(\mathsf{A}_{\mathsf{sync}}))}$. It now tries to restore the order in which 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 clock-out 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.

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 $\mathcal{I}_{\mathsf{p}_{\mathsf{A}_{\mathsf{sync}}}?=(i,j)}$ where i denotes the global round and j denotes the subround the adversary is clocked in.[6] 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

[6] The adversary obviously knows both i and j because he knows the clocking scheme κ, so he may simply maintain two counters that he adapts every time he is clocked.

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 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 can show the following lemma.

Lemma 5. *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 4. 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.

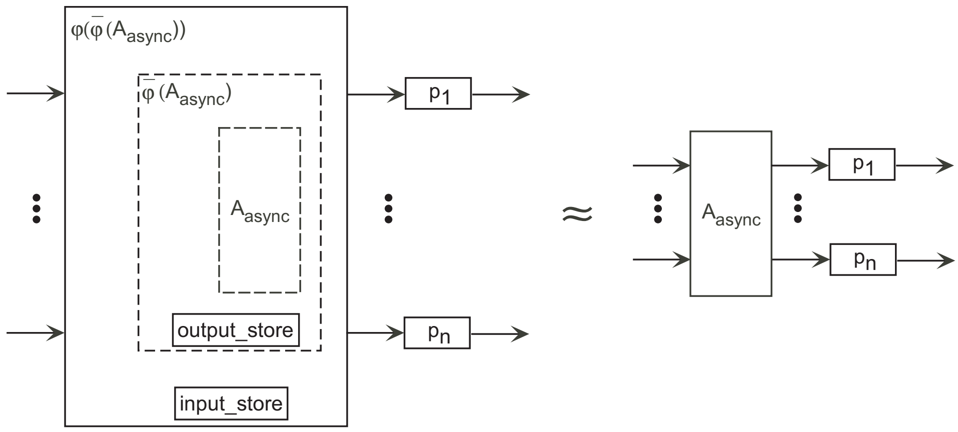

Fig. 4. Overview of the proof of Lemma 5.

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.[7] 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 $\mathsf{p}? \neq \mathsf{p}_{\mathsf{A}_{\mathsf{sync}}}?$. 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 to 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 $\widetilde{\mathsf{p}}$ at the same position, so they remain validly mapped. Now, either the first component of $output\_store_{\mathsf{p}!}$ or the first entry of $\widetilde{\mathsf{p}}$ 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 5, 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 1 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, 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

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

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

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

■

[7] 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 $\varphi_{conf}(\bar{\varphi}_{conf}(conf_{\mathsf{async}}))$.

Proof. (Theorem 4) Before we turn our attention to the actual proof, we state the following lemma which captures that we can "locally reverse" the function ϕ for the honest user:

Lemma 6. *Let a synchronous system Sys_{sync}, a clocking scheme κ and a 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. Let $conf_{\mathsf{async}} = (\varphi(\hat{M}_{\mathsf{sync}}) \cup \{\mathsf{X}_{\mathsf{sync},\kappa}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \mathsf{A}')$ be an arbitrary asynchronous configuration. If we now have given $\phi(view_{conf_{\mathsf{async}}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$ then we can "locally reverse" the function ϕ for the view of the user, i.e., we can define a function ϕ_{H}^{-1} on the runs of the synchronous configuration, such that*

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

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

Proof. (Lemma 6) 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 to 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 to the view of the honest user if it is used in a polynomial-time configuration. ■

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 2 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. Now the precondition $Sys_{\mathsf{sync},1} \geq_{\mathsf{sync}} Sys_{\mathsf{sync},2}$ can be applied yielding a configuration $conf_{\mathsf{sync},2} = (\hat{M}_{\mathsf{sync},2}, S, \mathsf{H}_{\mathsf{sync}}, \mathsf{A}_{\mathsf{async},2}) \in \mathsf{Conf}(Sys_{\mathsf{sync},2})$ with

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

   and $(\hat{M}_{\mathsf{sync},2}, S) \in f(\hat{M}_{\mathsf{sync},1}, S)$. Moreover, in the computational case, $conf_{\mathsf{sync},2}$ is polynomial-time.

3. We now apply Theorem 1 to the configuration $\mathit{conf}_{\mathsf{sync},2}$ which yields a configuration $\mathit{conf}_{\mathsf{async},2} = (\varphi(\hat{M}_{\mathsf{sync},2}) \cup \{\mathsf{X}_{\mathsf{sync},2,\kappa_2}\}, S, \varphi(\mathsf{H}_{\mathsf{sync}}), \varphi(\mathsf{A}_{\mathsf{sync},2}))$ with

$$\mathit{view}_{\mathit{conf}_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}}) = \phi(\mathit{view}_{\mathit{conf}_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$

   Moreover, $\mathit{conf}_{\mathsf{async},2}$ is a polynomial configuration iff $\mathit{conf}_{\mathsf{sync},2}$ is polynomial, according to Theorem 1.
4. Putting it all together, we have
   - $\phi(\mathit{view}_{\mathit{conf}_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) = \mathit{view}_{\mathit{conf}_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}})$
   - $\mathit{view}_{\mathit{conf}_{\mathsf{sync},1}}(\mathsf{H}_{\mathsf{sync}}) \approx \mathit{view}_{\mathit{conf}_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})$
   - $\mathit{view}_{\mathit{conf}_{\mathsf{sync},2}}(\mathsf{H}_{\mathsf{sync}})) = \phi(\mathit{view}_{\mathit{conf}_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}})))$

   Using Lemma 2, we obtain

$$\phi(\mathit{view}_{\mathit{conf}_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}}))) \approx \phi(\mathit{view}_{\mathit{conf}_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}}))).$$

   We now finally apply our "reversing" function ϕ_{H}^{-1} (cf. Lemma 6) on the above equation. Together with Lemma 2

$$\mathit{view}_{\mathit{conf}_{\mathsf{async},1}}(\varphi(\mathsf{H}_{\mathsf{sync}})) \approx \mathit{view}_{\mathit{conf}_{\mathsf{async},2}}(\varphi(\mathsf{H}_{\mathsf{sync}})).$$

   Hence, $\mathit{conf}_{\mathsf{async},2}$ is an indistinguishable configuration for $\mathit{conf}_{\mathsf{async},1}$. Moreover, we have $(\hat{M}_{\mathsf{sync},2}, S) \in f(\hat{M}_{\mathsf{sync},1}, S)$, i.e., $\varphi(\hat{M}_{\mathsf{sync},2}, S) \in f'(\varphi(\hat{M}_{\mathsf{sync},1}, S))$, which yields the desired result $\varphi(\mathit{Sys}_{\mathsf{sync},1}) \geq^{f'}_{\mathsf{async},\mathsf{H}} \varphi(\mathit{Sys}_{\mathsf{async},2})$.

■