Karlsruhe Reports in Informatics 2015,10
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Non-Interference with What-Declassification in Component-Based Systems

Daniel Grahl and Simon Greiner

2015

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Non-Interference with What-Declassification in Component-Based Systems

Daniel Grahl and Simon Greiner
Karlsruhe Institute of Technology, Germany
Department of Informatics
{firstname.lastname}@kit.edu

***Abstract*—Component-based design is a method for modular design of systems. The structure of component-based systems follows specific rules and single components make assumptions on the environment that they run in. In this paper, we provide a non-interference property for component-based systems that allows for a precise specification of what-declassification of information and takes assumptions on the environment into consideration in order to allow a modular, precise and re-usable information-flow analysis. For precise analysis, components can be analyzed by separately analysing services provided by a component, and from our compositionality theorem non-interference of components follows.**

I. Introduction

Modern IT Systems have to master complex requirements regarding functionality, availability, and scalability. Designing these kinds of systems as monolithic programs is not feasible in practice. Component-based system engineering techniques allow designers to break a system down to different, reusable parts specializing on particular tasks. Each of these components often do not provide a functionality on their own but only in cooperation with other components. Contracts allow single components to explicitly formulate requirements on their environment which are necessary in order to provide guarantees of a component's implementation.

In this work, components are programs encapsulating their states and communicating with their environment exclusively via messages. Functionality is provided as services, i.e., requests made from one component to another and an answer to this request. This notion is inspired by Szyperski et al. [1]. Real world implementations are provided especially in the web context, for example using Enterprise Java Beans [2], a common technology for implementing distributed, service-oriented systems in Java. Component-based Systems can be considered either as programs where different components run on distributed hardware, running in different virtual machines or, for example like web applications and databases, running in different processes on the same operating system.

The distribution of functionality to different components makes it hard to provide precise guarantees of information flow properties of an overall system. When information is exchanged between components, it is necessary to preserve knowledge about the sensitivity of the information passed along. In this paper, we develop a compositional non-interference property with what-declassification for component-based systems and we thus allow a very precise specification of high and low information on interfaces between components. Further, the non-interference property takes into consideration typical guarantees assumed by Szyperski-like components, for example, the existence of required services or termination of a service.

Figure 1 illustrates a component-based system of a web-based online shop. Component C receives the name and a credit card number from the customer via a web form. Then C stores this information in the database DB, from where the billing component B requests this information in order to generate a bill; and component P manages the payment process with the credit card company. It is necessary that the bill contains the customer's name, but the credit card number can be restricted to the last four digits. To perform the payment, however, the entire information has to be used by component P.

The information that only the last four digits of the credit card are low is specified on the input-level of C. In order to allow modularity, this information has to be preserved over the boundaries between components C and DB, as well as between DB and B.

A. Contribution

We introduce a compositional notion for non-interference in interactive programs supporting declassification of information. Our framework is an extension of the framework by Rafnsson et al. [3]. Our notion of declassification has as a consequence that non-interference in presence of deterministic environments is weaker than non-interference in general. Since equivalence of non-interference with deterministic environments and non-interference in general is the basis for compositionality proofs in [3], we provide a compositionality proof for non-interference with declassification.

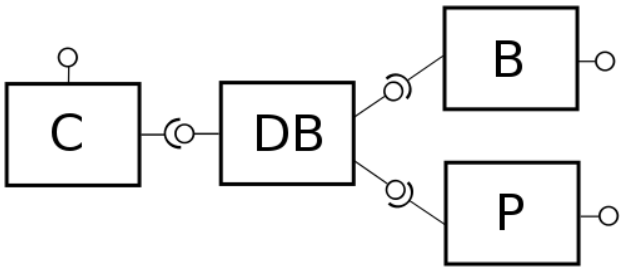

Figure 1. Component Diagram

We provide a formalization of components as sets of services operating on separated states making assumptions on the environment the component is executed in. The formalization of components is inspired by restrictions the Java Enterprise Edition makes on Java programs executed in application containers. This formalization should also be applicable to other web technologies like ASP.NET or databases. We formalize non-interference for components in the presence of cooperative environments and we show compositionality for synchronous parallel composition.

In general, it is not possible to analyze non-interference with declassification for components using approaches like type systems and others. Therefore, we aim for static verification approaches of non-interference properties in components based on theorem provers. To achieve this, we introduce a notion of non-interference for services, which implies non-interference for components and is suitable for extensions of theorem prover approaches.

B. Outline

In the following section, we introduce the computational model we use, based on labeled transition systems. In Section III, non-interference is defined and we show parallel interleaving compositionality. This is followed by the definition of components in Section IV and the definition of non-interference for components in Section V.

After this, we introduce the computational model we use, based on labeled transition systems. In Section III, we extend strategy-based non-interference with what-declassification using equivalence relations for specification. For better comparability with the original work by Rafnsson et al. we make this extension on their original computational model, provide a comparison with their compositionality results and present a proof for compositionality of our extended notion. In Section IV, we define components and compositions, our target of analysis in this paper. This is followed by the definition of non-interference for components under cooperative environments in Section V. Finally, we provide a non-interference property for services, which implies non-interference for components and is suitable for static program analysis. In Section VII we present related work and finally conclude.

II. Computational Model

In this work, we discuss non-interference in component-based systems. A core property of components in our setting is that components do not share a state and thus can not communicate via variable manipulation. The only form of communication is via messages sent and received by a component. Labeled Transition Systems (LTS) are a very general possibility for modeling programs or components communicating via message passing.

The environment communicates with an LTS by passing messages over channels and observing output messages from the LTS. We define $\mathbb{C}$ as a set of channels, over which values from a domain $\mathbb{V}$ can be communicated. We refer to the communication of a value over a channel as a message in $\mathbb{M} \subseteq \mathbb{C} \times \mathbb{V}$. An LTS can receive input messages $\mathbb{I}$ and send output messages $\mathbb{O}$, where $\mathbb{M} = \mathbb{I} \cup \mathbb{O}$ and $\mathbb{I} \cap \mathbb{O} = \emptyset$. We write $\alpha!v$ for a message $m \in \mathbb{O}$ communicating the value v on a channel α, $\alpha?v$ for $m \in \mathbb{I}$, and if it is not relevant whether m is an input or output, we write $\alpha.v$ for $m \in \mathbb{M}$. The transition relation $\rightarrow$ describes the transition of an LTS by communicating a message. We write $p \xrightarrow{m} p'$, if LTS p transitions to p' for some $m \in \mathbb{M}$. We write $p \xrightarrow{m}$ if there exists some p' such that $p \xrightarrow{m} p'$.

Further, we require an LTS not to discriminate on input acceptance due to the communicated value (1) and neither to provide indeterministic output (2) nor indeterministic internal behavior (3).

1) $\forall \alpha \in \mathbb{C}, v \in \mathbb{V} \cdot p \xrightarrow{\alpha?v} \implies \forall v' \in \mathbb{V} \cdot p \xrightarrow{\alpha?v'}$
2) If $p \xrightarrow{m_1} p_1$ and $p \xrightarrow{m_2} p_2$ and $m_1 \neq m_2$ then $m_1 \in \mathbb{I}$ and $m_2 \in \mathbb{I}$.
3) If $p \xrightarrow{m} p_1$ and $p \xrightarrow{m} p_2$ then $p_1 = p_2$.

The only source of indeterministic behavior is due to indeterministic input. In related work [3] this restricted form of an LTS is called "input-output LTS". For simplicity, we directly refer to them as LTS in the remainder.

Traces $\mathbb{T}$ are finite lists of messages, where $\langle\rangle \in \mathbb{T}$ is the empty list and $\frown$ is the concatenation operator. The trace consisting of a single message m is $\langle m \rangle$. If it is clear from the context we write m instead of $\langle m \rangle$. The prefix relation is defined as $t_1 \leq t$ if $\exists t_2 \cdot t_1 \frown t_2 = t$. We say p can *communicate* a trace t, written $p \xrightarrow{t}$, if $t = \langle\rangle$ or $t = \langle m \rangle \frown t'$ and $p \xrightarrow{m} p'$ and $p' \xrightarrow{t'}$ for some p'. $\mathbb{T}(p) := \{t \mid p \xrightarrow{t}\}$ is the set of all traces, which p can communicate.

For better readability, we frequently use the following convention in this paper. We refer to traces by t, channels by α and β, messages by m and values by v, and LTS by p as well as their primed and indexed counterparts.

III. Information-Flow in Labeled Transition Systems

Intuitively, non-interference for an LTS means, that some secret (high) information given as input to an LTS does not influence publicly (low) observable output of the LTS. To analyze information flow for an LTS a specification of secret and public information is necessary. Typically, this specification is, depending on the framework, given in the form of types of variables, parameters or channels. For example, if some channel has high type, the behavior of the LTS should not reveal the value of the communicated value. The behaviour of the LTS has to be *equivalent* for different values communicated over a high parameter.

For a compact presentation, we only consider the 2-element security lattice $high$ and low in this paper. Nevertheless, a more complicated security lattice can easily be expressed with our notion of security, but explicit consideration does not provide further insights.

Specification with types is very coarse-grained, since it does not allow to specify that only partial information contained in an input may be public (What-declassification [4]). For example, it

can not be expressed that the last bit of the input value may be revealed, but otherwise the value has to be kept secret. In this paper we introduce a specification of high and low information based on equivalence relations. If two messages are equivalent with respect to this specification, the observable behavior of the LTS should be indistinguishable for an adversary. This is a generalization of specification using types, and allows a very flexible specification of secret information, but we can express typed-based secrecy specifications with our relation.

A. Security Specification of Messages and Values

We assume the classification of high and low input and output for a labeled transition system is provided by an equivalence relation $\sim \subseteq \mathbb{M} \times \mathbb{M}$ as part of the specification. If two messages m_1, m_2 are equivalent with respect to $\sim$, the information which discriminates m_1 from m_2 is secret. For example, given the security specification $\alpha?x \sim \alpha?y$ iff $x \ mod \ 2 = y \ mod \ 2$, the last bit of the value communicated on channel α is low. The specification $\alpha?x \sim \alpha?y$ iff $x = y$, the entire communicated value is low and in the case $\alpha?x \sim \alpha?y$ iff $true$, the entire value is high. We assume in the reminder of this paper some definition of $\sim$ to be given.

In order to specify that the existence of a message itself is a secret, we introduce a special placeholder $\Box \in \mathbb{M}$. We call a message $m \sim \Box$ *invisible*, and *visible* if $m \nsim \Box$. If a message m is invisible, the observable behavior of an LTS must not differ depending on whether m is provided as an input or not.

We assume that it always can be distinguished on which channel the message was communicated, if the message is visible. Formally, this means $m \sim m' \implies (m \sim \Box \wedge m' \sim \Box) \vee (m = \alpha.v \wedge m' = \alpha.v')$ for some $\alpha \in \mathbb{C}$.

The equivalence relation $\sim$ implicitly defines *equivalence classes* on $\mathbb{M}$ with $[m] := \{m' \mid m' \sim m\}$. For every equivalence class $[m]$, we denote an arbitrary, but constant *representative* $[\![m]\!] \in [m]$, where $[\![\Box]\!] = \Box$.

Equivalence of messages gives rise to the equivalence of traces t, t', written $t \sim t'$. Traces t, t' are equivalent, if, after removing invisible messages, their *projection* on the representative of the equivalence classes are equal. We define $t \sim t'$ if $t\restriction_\sim = t'\restriction_\sim$ where

$$\langle\rangle\restriction_\sim := \langle\rangle$$
$$(m \frown t)\restriction_\sim := \begin{cases} t\restriction_\sim & \text{if } m \sim \Box \\ [\![m]\!] \frown t\restriction_\sim & \text{otherwise} \end{cases}$$

While above we introduced $\sim$ for messages, we overload the symbol for equivalence of traces in order to avoid too many different symbols. It should be clear from the context, whether $\sim$ refers to messages, traces, or as defined later, for sets of messages.

Apart from the projection operator on traces, we define projection on sets and a *filter* operator on traces and sets. Let $M, N \subseteq \mathbb{M}, m \in \mathbb{M}, t \in \mathbb{T}$.

- $M\restriction_\sim := \{[\![m]\!] \mid m \in M\} \setminus \{[\Box]\}$
- $M \sim N :\Leftrightarrow M\restriction_\sim = N\restriction_\sim$
- $M \rhd N := M \cap N$
- $\langle\rangle \rhd N := \langle\rangle$
- $(m \frown t) \rhd N := \begin{cases} m \frown (t \rhd N) \text{ if } m \in N \\ t \rhd N \text{ otherwise} \end{cases}$

B. Strategies

The environment observes the behavior of an LTS and provides input depending on this observation. The environment may also deny to provide further input. We model the environment as a *strategy*, a function mapping the previously communicated trace, i.e. the observation made by the environment to the LTS, to a set of possible inputs provided by the environment.

Our goal is to analyze non-interference for LTS, meaning we want to detect leaks in the LTS. When analyzing the LTS in the presence of an environment, we want to ensure that detected leaks are due to an insecurity in the LTS rather than an environment which leaks confidential information. To enforce the environment not to leak secret information, we require the strategy to provide equivalent input for equivalent observations (Line 1 in Definition 1). We denote the set of all strategies with `Strat`.

Definition 1 (Strategy)**.** *A* strategy *is a function* $\omega : \mathbb{T} \mapsto \mathcal{P}(\mathbb{I})$*, such that*

$$t_1 \sim t_2 \implies \omega(t_1) \sim \omega(t_2) \tag{1}$$

for all t_1, t_2*.*

A trace t is *consistent* with a strategy ω, written $\omega \models t$, if the strategy can communicate the trace. Formally $\omega \models t$, if $\forall m \in \mathbb{I}$ with $t = t_1 \frown m \frown t_2$ it holds that $m \in \omega(t_1)$. An LTS p produces t under ω, written $\omega \models p \xrightarrow{t}$ if ω is consistent with t and $p \xrightarrow{t}$. Again, this definition of strategies is a generalization of the previous work in [3].

If a strategy ω provides at most the input another strategy ω' provides, we say ω *refines* ω'.

Definition 2 (Strategy Refinement)**.** ω refines ω'*, written* $\omega \leq \omega'$*, if* $\omega(t) \subseteq \omega'(t)$ *for all* t*.*

A strategy ω refining ω' is at most consistent with the traces ω' is consistent with.

Lemma 1. *If* $\omega \leq \omega'$ *then for all LTS* p*:* $\omega \models p \xrightarrow{t} \implies \omega' \models p \xrightarrow{t}$

Proof for Lemma 1. Proof according to [5]. In general, every trace accepted by ω is also accepted by ω', due to definition of strategy acceptance and refinement relation. $\omega \models t$, so for every $m \in \mathbb{I}$: $t' \frown m \leq t \implies m \in \omega(t')$ and since $\omega(t') \subseteq \omega'(t')$ $m \in \omega'(t')$, so $\omega' \models t$. ◊

Two strategies are equivalent with respect to $\sim$, if they provide equivalent output for the same observation.

Definition 3 (Equivalence of strategies)**.** *Two strategies* ω *and* ω' *are* equivalent *with respect to an equivalence relation* $\sim \subseteq \mathbb{M} \times \mathbb{M}$ *if* $\forall t \in \mathbb{T} \cdot \ \omega(t)\restriction_\sim = \ \omega'(t)\restriction_\sim$*.*

Again, we overload here the symbol $\sim$ and write $\omega \sim \omega'$ for strategies that are equivalent with respect to $\sim$.

C. Non-Interference

After setting up a formal notion for LTS and a model for the environment, we can formalize non-interference. As usual, we compare different runs of an LTS. Since execution of an LTS in general requires intermediate input, the runs have to be executed in presence of an environment. We want to ensure that different behavior of runs is due to leaks in the LTS, not by the environment leaking secrets. Therefore, we require the different runs to be executed under equivalent environments. We say an LTS is non-interferent, if every trace which is possible for the LTS under a strategy is also possible under every other, equivalent strategy.

An LTS p is non-interferent with respect to a set of strategies W, if for every trace produced by p under a strategy $\omega \in W$, an equivalent trace is produced under every other, equivalent strategy in W.

Definition 4 (W-non-interference)**.** *An LTS p is W-non-interfering for $W \subseteq \texttt{Strat}$, if*

$$\forall \omega_1, \omega_2 \in W, t_1 \cdot \omega_1 \sim \omega_2 \ \wedge \ \omega_1 \models p \xrightarrow{t_1} \Longrightarrow$$
$$\exists t_2 \cdot \omega_2 \models p \xrightarrow{t_2} \wedge \ t_1 \sim t_2$$

A *W-attack* is a counter example for W-non-interference.

Definition 5 (Attack)**.** *A W-*attack *on p is a tuple $(\omega_1, \omega_2, t_1) \in W \times W \times \mathbb{T}$ with*

1) $\omega_1 \sim \omega_2$ *and*
2) $\omega_1 \models t_1$ *and*
3) $\omega_1 \models p \xrightarrow{t_1}$ *and*
4) $\forall t_2 \cdot \omega_2 \models p \xrightarrow{t_2} \Longrightarrow \ (t_1 \nsim t_2)$

It is easy to see that an LTS is W-non-interferent if and only if there does not exist a W-attack.

We denote the set of all programs, which are W-non-interferent with W-NI.

If a program is W-non-interferent, it is also non-interferent with respect to all subsets of W.

Lemma 2. *For all $W_1, W_2 \subseteq \texttt{Strat}$: $W_1 \subseteq W_2 \implies W_2$-NI $\subseteq W_1$-NI*

Proof for Lemma 2. We have to proof that, given $W_1 \subseteq W_2$, all $p \in W_2$-NI it also holds that $p \in W_1$-NI. We show the contrapositive. Assume $p \notin W_1$-NI. Then there exists (ω_1, ω_2, t) which is a W_1-attack on p. Since $W_1 \subseteq W_2$, $\omega_1 \in W_2$ and $\omega_2 \in W_2$, (ω_1, ω_2, t) is a W_2-attack on p. $\Diamond$

D. Non-interference for Deterministic Strategies

We have presented a notion of non-interference, which is a generalization of previous work and adds what-declassification. In this section we show how our non-interference with declassification relates to some properties of non-interference *without* declassification. One property of non-interference is that it is sufficient to only consider deterministic strategies in order to prove non-interference with respect to all possible strategies. This finding also is the basis for all proofs of compositionality properties in [3]. We show here that this property does not hold if we allow declassification.

A strategy is deterministic, if it provides for any channel at most one message for every observation.

Definition 6 (Deterministic Strategies)**.** *A strategy ω is* deterministic *if* $\forall \alpha, t \cdot |\{m \mid m \in \omega(t) \wedge m = \alpha?v\}| \leq 1$.

We refer to the set of all deterministic strategies as DS.

Theorem 1. *Strat-NI $\subsetneq$ DS-NI*

To illustrate Theorem 1 consider the specification $\alpha?x \sim \alpha?y$ if $x \neq 1 \wedge y \neq 1$, $\alpha?1 \sim \Box$ and $\beta!x \sim \beta!y \Leftrightarrow x = y$.

This specification expresses that the only information that may be leaked on channel α is whether the communicated value is equal to 1. Further, if the message communicates 1, then the message is invisible. Also, $\beta!x \sim \beta!y \Leftrightarrow x = y$, expressing that every information communicated over β can be distinguished.

The program $\texttt{read}(x \leftarrow \alpha); \texttt{read}(x \leftarrow \alpha); \texttt{write}(1 \rightarrow \beta)$ is non-interferent for deterministic strategies, but not for all strategies. Take for example the trace $\alpha?1; \alpha?2; \beta!1$. This trace reveals the existence of $\alpha?1$ and we can obviously construct a Strat-Attack.

But we can not construct a deterministic strategy accepting this trace. For a strategy ω accepting the trace, it has to hold that $\alpha?1 \in \omega(\langle\rangle)$ and $\alpha?2 \in \omega(\langle\alpha?1\rangle)$. Since $\langle\rangle \sim \langle\alpha?1\rangle$, also $\alpha?x \in \omega(\langle\rangle)$ for some $x \neq 1$. And therefore ω is not deterministic according to Definition 6.

Proof for Theorem 1. Strat-NI $\subseteq$ DS-NI follows from Lemma 2. As a counterexample for Strat-NI $=$ DS-NI see example above. $\Diamond$

The reason for Strat-NI $\neq$ DS-NI in our generalized case, when $\sim$ defines declassification, is that determinism is defined on the level of channels, while equivalence of messages can imply several equivalence classes for messages communicated over one channel. The specification of high information in the specific case of non-interference by Rafnsson et al. ensures that either all messages communicated over a channel are equivalent or none are. Especially, there can not exist two messages on one channel, where one message on some channel is visible, and the another message on the same channel is not. This limitation on the security specification does not apply in our case as the counterexample above illustrates.

E. Parallel Compositionality

The most general composition of LTS is by parallel interleaving of LTS. In parallel interleaving compositions, composed LTS do not directly communicate with each other, but the output of one LTS is mapped to input of the other LTS by the environment. The environment has full control over the wiring between the composed LTS. The semantics of parallel composition is defined in Figure 2.

Rafnsson et al. show that non-interferent LTS result in a non-interferent LTS when parallelly composed. Their proof makes use of the property Strat-NI $=$ DS-NI, which holds when $\sim$ does not define declassification. But as we have shown in Theorem 1, Strat-NI $=$ DS-NI does not hold in our

$$\frac{p_1 \xrightarrow{m} p_1'}{(p_1 \parallel p_2) \xrightarrow{m} (p_1' \parallel p_2)} \qquad \frac{p_2 \xrightarrow{m} p_2'}{(p_1 \parallel p_2) \xrightarrow{m} (p_1 \parallel p_2')}$$

Figure 2. Parallel composition of LTS

generalized case. Nevertheless, the compositionality of LTS still holds.

Theorem 2 (Compositionality). $p_A, p_B \in \texttt{Strat-NI} \implies p_A \parallel p_B \in \texttt{Strat-NI}$

The proof for Theorem 2 requires some additional definitions. Therefore, we refer the interested reader to the appendix.

IV. Components and Compositions

We have defined a non-interference property allowing the declassification of partial information. Now, we want to use this property to construct systems using a component-based system design. Component-based system development allows efficient design of systems and makes use of re-usability of parts of a system. To allow re-usability, including non-interference properties, we require a black-box view on information flow in a component. We consider components to be some entity which provides services to its environment and requires other services to be provided by the environment. Services provided by one component share a common state, while a component's state is disjoint from all other components.

The correct functionality of a component depends on guarantees the environment has to provide. One of these guarantees is that the environment provides services necessary for the correct execution of a component's services. A component guarantees to provide services, if it can rely on the environment providing a set of required services.

The concrete formalization of components is inspired by the Java Enterprise Edition (JEE) as specified in the JSR 318, version 3.1 [2], a technology for implementing business logic on distributed server systems. Business logic in JEE is implemented in so called *Java Beans*, Java objects, which are run by an Application Server and whose methods can be called as services by remote calls. The programming model is rather restrictive in order to allow the application server to perform tasks like login, session management, and thread management.

JEE mainly differentiates between stateless and stateful session beans. Stateless session beans do not manage a state over the execution of a service, while stateful beans preserve their state over several sequent service calls. JEE requires that beans may at most be entered by a single thread, which by construction of JEE applications means that beans are not reentrant and loop-back calls are not allowed. This enforces a strict layering of JEE applications, since circular dependencies might result in callback loops. Service calls are synchronous in the sense that after the call of a service in another bean the execution of the calling service waits for termination of the called service. Asynchronous service calls can be performed by message driven beans, but we do not consider message driven beans here for simplicity.

In this section, we formalize components and services. Also, we define sets of strategies, which respect assumptions the component makes about the environment. We call these environments *cooperative*. Then we define a composition of components which allows the reduction of assumptions a component makes about the environment. In this setting, we show non-interference of compositions under cooperative environments. Ultimately, we show that a component-based system is in Strat-NI, if it does not require any services to be provided by the environment.

A. Services and Components

The environment can call a *service* by sending a message to the component. During execution of the service, the component can call other services provided by other components by sending and receiving messages. Finally, after termination of the service, some return value is provided as a response to the initial call. A service is defined by a name, a signature and the body of the program defining the service's behavior. The signature describes the input parameters of the service, the initial channel used for calling the service, and the termination channel used for communicating the return value. A combination of the program defined by their services of a component and a state, as defined later, define an LTS. The function $Ini(serv)$ defines the *initial channel* of the service $serv$, $Fin(serv)$ the *termination channel*.

The *body* of a service is a program consisting of the language primitives from Figure 3. We give the semantics of the language with respect to some state σ. A state is a mapping from program variables to values from the domain $\mathbb{V}$. In the remainder of this paper, σ and its primed and indexed counterparts refer to states.

For simplicity, we do not define methods. Also, we refrain from an explicit consideration of objects. The language could easily be extended to support both of them without invalidating the results in the remainder, but it would complicate the presentation without any considerable benefit.

The rule $Service$ in Figure 3 defines the semantics of a service call. A service call consists of the sending of a message on the initial channel and providing some value representing a parameter. After sending, the service waits for the response of the called service on the termination channel. Semantics of sending and receiving messages is shown in Figure 4. We refer to the body of a service $serv$ by $body_{serv}$. Note that by definition of the language, services are deterministic.

The services defined by the language introduced above are limited to a single parameter. This limitation is not a restriction of the expressivity of our language. The parameter can be considered to be some encoding of several parameters. As illustrated in the introduction, secrecy of information contained in the parameter can be modeled with $\sim$ for each parameter separately, also for parts of the parameter or combinations of several parameters included in one service call. We limit the presentation here to a single parameter, since it simplifies the presentation without restricting the results.

The handler of a service represents the program which is executed when a service is called. Initially, the service is started by a message on channel $Ini(serv)$ and after executing the service's body, it writes the result on channel $Fin(serv)$. We assume variables named $param$ and res to be available in every component, the variable named $param$ can be used by the program to access the parameter and res to write the return value. The program representing the handler $handler_{serv}$ of a service $serv$ is defined as

$$handler_{serv} := \texttt{read}\,(param \leftarrow Ini(serv))\,;body_{serv};\ \texttt{write}\,(res \rightarrow Fin(serv))\,;$$

$$\textsc{Skip}\ \frac{}{\langle SKIP;\sigma\rangle \rightarrow \langle SKIP;\sigma\rangle}$$

$$\textsc{Seq1}\ \frac{}{\langle SKIP;c_2;\sigma\rangle \rightarrow \langle c_2;\sigma\rangle}$$

$$\textsc{Seq2}\ \frac{\langle c_1;\sigma\rangle \xrightarrow{t} \langle c_1';\sigma'\rangle \quad c_1 \neq SKIP}{\langle c_1;c_2;\sigma\rangle \xrightarrow{t} \langle c_1';c_2;\sigma'\rangle}$$

$$\textsc{Assign}\ \frac{\sigma(e)=v}{\langle x:=e;\sigma\rangle \rightarrow \langle SKIP;\sigma[x:=\sigma(e)]\rangle}$$

$$\textsc{If1}\ \frac{\sigma(e)=0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2;\sigma\rangle \rightarrow \langle c_1;\sigma\rangle}$$

$$\textsc{If2}\ \frac{\sigma(e)\neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2;\sigma\rangle \rightarrow \langle c_2;\sigma\rangle}$$

$$\textsc{While}\ \frac{}{\langle \texttt{while}\ e\ \texttt{do}\ c_1;\sigma\rangle \rightarrow \langle \texttt{if}\ e\ \texttt{then}\ (c_1;\texttt{while}\ e\ \texttt{do}\ c_1)\ \texttt{else}\ SKIP;\sigma\rangle}$$

$$\textsc{Service}\ \frac{servC = Ini(serv) \quad servR = Fin(serv)}{\langle x := \text{serv}(e);\sigma\rangle \rightarrow \langle \texttt{write}\,(e \rightarrow servC)\,;\texttt{read}\,(x \leftarrow servR)\,;\sigma\rangle}$$

Figure 3. Language Semantics

$$\textsc{Send}\ \frac{\sigma(e)=v}{\langle \texttt{write}\,(e \rightarrow \alpha)\,;\sigma\rangle \xrightarrow{\alpha!v} \langle SKIP;\sigma\rangle}$$

$$\textsc{Rec}\ \frac{e \in \mathbb{V}}{\langle \texttt{read}\,(x \leftarrow \alpha)\,;\sigma\rangle \xrightarrow{\alpha?e} \langle SKIP;\sigma[x:=e]\rangle}$$

Figure 4. Additional semantic rules

$$\textsc{Ext1}\ \frac{\langle c_1;\sigma\rangle \xrightarrow{\alpha?x} \langle c_1';\sigma'\rangle}{\langle c_1 \sqcap c_2;\sigma\rangle \xrightarrow{\alpha?x} \langle c_1';\sigma'\rangle}$$

$$\textsc{Ext2}\ \frac{\langle c_2;\sigma\rangle \xrightarrow{\alpha?x} \langle c_2';\sigma'\rangle}{\langle c_1 \sqcap c_2;\sigma\rangle \xrightarrow{\alpha?x} \langle c_2';\sigma'\rangle}$$

Figure 5. Semantics of external choice

A service $serv_1$ *requires* a service $serv_2$, if a call to $serv_2$ is contained in the body of $serv_1$. We denote the set of services required by $serv_1$ with $\texttt{req}_{serv_1}$.

A component c provides a set of services $\texttt{prov}_c$ to its environment. We recursively define the body of the component, refered to as c_{body}, with respect to the services it provides. The component initially provides all services, while the environment chooses which service should be executed by sending a message to the respective initial channel. After termination of the called service, again, the environment can choose among all provided services. The *external choice* operator $\sqcap$ models this behavior. Semantics of $\sqcap$ is shown in Figure 5.

Let $\{serv_1, ..., serv_n\} = \texttt{prov}_c$ for some component c. Then the body of c is recursively defined as

$$c_{body} := (handler_{serv_1} \sqcap \ldots \sqcap handler_{serv_n}); c_{body}$$

We assume for every component c to have some unique initial state σ_c without explicitly specifying it. $\langle c_{body};\sigma_c\rangle$ represents an LTS as defined earlier. We refer to this LTS with c_{LTS}.

Services provided by one component share a common state, i.e. information received by one service might be leaked by another, subsequently executed, service. Further, we assume that services are executed sequentially. We require services to terminate by definition, in contrast a component can never terminate. After successful execution of a service, all services are again offered to the environment.

The set of services that a component requires is the union of the required services of the provided services, i.e. $\texttt{req}_c = \{serv \mid \exists s{\cdot}\, s \in \texttt{prov}_c \wedge serv \in \texttt{req}_s\}$

If a service is called with an invisible message, but terminates with a visible message, the termination of a service reveals the initial message. So, it makes sense to require services not reveal its call by a visible termination message. For the same argument, we make sure that the call of a service does not reveal the upcoming termination of the service.

Definition 7 (Visibility-Preserving services)**.** *A service* $serv$ *is* visibility-preserving *if* $\forall \sigma, \sigma', v, v', t{\cdot}$ $\langle handler_{serv};\sigma\rangle \xrightarrow{Ini(serv)?v \rightsquigarrow t \rightsquigarrow Fin(serv)!v'} \langle SKIP;\sigma'\rangle \implies (Ini(serv)?v \sim \Box \Leftrightarrow Fin(serv)!v' \sim \Box)$

A component c *is* visibility-preserving *if all services in* $\texttt{prov}_c$ *are* visibility-preserving.

The contract between a component and its environment states that the component only guarantees correct execution if required services are provided by the environment. Since

components are designed for compositional system design, a component also has to work as part of the environment to other components. Therefore, we have to ensure that services provided by a component terminate and all provided services can be called again. Termination, as stated above, requires that a service ensures that for every trace a service can communicate, there exists a trace which leads to termination of the service.

Definition 8 (Terminating Service)**.** *A service $serv$ is* terminating *if*

$$\forall t, \sigma\ \exists t' \cdot \langle serv_{handler}; \sigma\rangle \xrightarrow{t} \Longrightarrow \langle serv_{handler}; \sigma\rangle \xrightarrow{t \frown t'} \langle SKIP; \sigma'\rangle \ \textit{and}$$
$$\forall \sigma \exists n \in \mathbb{N}\ \ \forall t \cdot \langle serv_{handler}; \sigma\rangle \xrightarrow{t} \Longrightarrow |t| \leq n$$

The first condition in Definition 8 ensures that every input provided by the environment leads to a state, where the service still can run to completion. The second condition ensures that the maximum length of a trace needed for completion only depends on the initial state, but not on intermediate input. It is possible to implement services which comply with the first part of the conjunction of this definition, but can still be forced by the environment to perform infinite execution. Take for example the following service:

$$x = \text{m}(0); \texttt{while}\ \ x == 1\ \ \texttt{do}\ \ x = \text{m}(0); return\ 1;$$

For every trace, the service can communicate, there exists the case, when the environment provides 0 as a result of m. Therefore, we additionally require an upper bound for the length of a trace a service can communicate. In combination with the two conditions, the implementation of the service has to ensure that it terminates, independent from the environment the component runs in.

For technical reasons, we assume that a service is at most provided by one component, i.e. $serv \in \texttt{prov}_c \wedge serv \in \texttt{prov}_d \implies c = d$. This restriction is useful in the further presentation, but limits the expressivity of our language only marginally. If two components are designed to provide the same service, one of them can be changed by a simple renaming of the initial and terminating channels and the name of the service. It does however imply a static system in the sense that we do not allow exchanging components at run-time.

B. Composition

The contract between a component and its environment implied by required and provided services states that correct functionality of a component depends on an environment providing some basic guarantees to the component. We call these environments *cooperative*. Only if a component does not require any services, the environment may be non-cooperative. When composing components interleaving parallel, i.e., the environment is responsible for message passing as defined in Subsection III-E, this dependency can not be mitigated. Nevertheless, we want to have a composition operation for components, which allows to reduce dependency of the composition from the environment.

$$\textsc{parsynch1}\ \frac{p \xrightarrow{\alpha.v} p' \wedge \alpha \notin C}{p[\![C]\!]s \xrightarrow{\alpha.v} p'[\![C]\!]s}$$

$$\textsc{parsynch2}\ \frac{s \xrightarrow{\alpha.v} s' \wedge \alpha \notin C}{p[\![C]\!]s \xrightarrow{\alpha.v} p[\![C]\!]s'}$$

$$\textsc{parsynch3}\ \frac{p \xrightarrow{\alpha!v} p' \wedge s \xrightarrow{\alpha?v} s' \wedge \alpha \in C}{p[\![C]\!]s \xrightarrow{\alpha!v} p'[\![C]\!]s'}$$

$$\textsc{parsynch4}\ \frac{p \xrightarrow{\alpha?v} p' \wedge s \xrightarrow{\alpha!v} s' \wedge \alpha \in C}{p[\![C]\!]s \xrightarrow{\alpha!v} p'[\![C]\!]s'}$$

Figure 6. Inference rules for synchronized parallel composition

We define a composition operation which reduces dependency on a cooperative environment using synchronized parallel composition. Parallelly synchronized composition for two LTS p and p' on a set of channels c, written $p[\![c]\!]p'$, means that communication between p and p' on some channel from c is performed by the components directly without utilizing the environment. The semantics of $p[\![c]\!]p'$ is defined in Figure 6. The communication between two components can still be observed by the environment, but is not provided by the environment. If two components are composed synchronously on the initial and terminating channels of a service, we remove this service from the set of required services.

We call the combination of two components using synchronized parallel composition a *composition*. Composed components communicate on a well-defined set of services by synchronizing on the respective initial and terminating channels. The environment is not involved in the message passing, but can still observe the internal communication. We provide a formal definition for components recursively.

Definition 9 (Composition)**.** *A component is a* composition*. For components (or compositions) c, c' with $\texttt{prov}_c \cap \texttt{prov}_{c'} = \emptyset$, $\texttt{req}_c \cap \texttt{req}_{c'} = \emptyset$, and $s \subseteq \texttt{req}_c \cap \texttt{prov}_{c'}$, d is the composition of c and c' on the set of services s, written $d := c[\![s]\!]c'$ and*

- $\texttt{prov}_d := \texttt{prov}_c \cup (\texttt{prov}_{c'} \setminus s)$
- $\texttt{req}_d := (\texttt{req}_c \setminus s) \cup \texttt{req}_{c'}$
- $d_{LTS} := c_{LTS}[\![Ini(s) \cup Fin(s)]\!]c'_{LTS}$

To compose two compositions on a set of services, one composition has to provide the services, while the other composition has to require them. The set of provided services of the composition results from the provided services of each component, minus the services the components synchronize on. Also, the set of required services is the combination of the services required by each component, minus the services provided internally. The LTS defined by the composition results from the parallel synchronous composition of the two LTS of the components.

Note we ensure that the channels used for synchronization represent calls and termination of required services for one component and provided services for the other component. This way, we enforce an acyclic structure in compositions, which guarantees us that there are no deadlock situations caused by one component being called, but the other component not being able to provide the called service.

Also, the states of each component can not be interfered with by the other component. The only way of interaction between the components in a composition is due to message passing.

By composition, services are removed from the set of required services, and the dependency on a cooperative environment can be reduced. Basically, we ensure that an assumption a component makes on a cooperative environment is provided by a component which is known and can be analyzed.

In a composition, formerly provided services are also removed from the set of provided services, but this is a mere technicality. We can always just add a copy of the respective services with renamed initial and terminating channels to the component.

C. Cooperative Strategies

We have modeled components and defined a composition operation for components. Also, we frequently state that environments have to be cooperative. We now define cooperative environments as a subset of strategies.

An environment is cooperative, if it satisfies three conditions.

1) Every service required by a component is provided by the environment.
2) Every service called by a component terminates.
3) A service is terminated with a visible message if it was called with a visible message.

Condition (1) is trivially satisfied by any strategy since the call of a service is an output message sent by the composition and strategies can not refuse outputs. Condition (2) is a real restriction on strategies. It ensures that if a composition calls a required service, the environment provides a message on the terminating channel. The last condition (3) is a condition which results from our non-interference definition. Assume a component would call a service with an invisible message. If the environment answers this call with a visible message, the environment leaks the information that the service was called. Since we do not consider information leaks caused by the environment, we rule out this leak by definition.

We call a strategy satisfying conditions (2) and (3) a *Cooperative Strategy*.

Definition 10 (Cooperative Strategies)**.** *Given composition c providing the services $\mathtt{prov}_c$, $\omega \in \mathtt{Strat}$ is a* cooperative strategy *for c, written $\omega \in \mathtt{COOP}_c$, if for all $t, t', serv, \sigma, v$ such that $serv \in \mathtt{req}_c$ and $\omega \models \langle c_{body}; \sigma\rangle \xrightarrow{t \frown Ini(serv)!v \frown t'}$ and $Fin(serv) \notin t'$, it holds*

$$\exists t'' \cdot \omega \models \langle c_{body}; \sigma\rangle \xrightarrow{t \frown Ini(serv)!v \frown t' \frown t''} \wedge \quad (2)$$

$$Fin(serv)?v \in \omega(t \frown Ini(serv)!v \frown t' \frown t'') \quad (3)$$

and

$$Fin(serv)?v' \in \omega(t \frown Ini(serv)!v \frown t') \quad (4)$$

$$\implies \quad (5)$$

$$Ini(serv)!v \sim \Box \Leftrightarrow Fin(serv)?v' \sim \Box \quad (6)$$

With the first restriction in Definition 10, we ensure that for every trace which is consistent with a cooperative strategy and a composition, which contains the call of a service, there also exists a trace (line 2) after which the called service terminates (line 3). For components, the termination has to be communicated right after the call of the service, because by construction of components, no other traces are accepted by the component. Intuitively, this restriction formulates that an attacker knows the cooperative. A strategy can not block execution by not providing invisible termination messages. So it is not a secret that termination messages are provided, given that a service was called. As a consequence, it may be a secret, whether the environment calls a service, but not whether a called service terminates.

The second restriction formalizes visibility preserving execution of services required by the component (Condition 3). It ensures that, if a cooperative strategy provides a terminating message for a trace, which contains the initial message for the service (line 4), then this terminating message is visible if and only if the initial message was also visible (line 6). This way, we avoid that the strategy leaks the information that an invisible service call happened by revealing the call through the termination message.

The set of cooperative strategies for a component or composition c indeed only limits the set of strategies, if the composition requires services and therefore expects a cooperative environment.

Lemma 3. *For a composition c with $\mathtt{req}_c = \emptyset$ it holds that $\mathtt{COOP}_c = \mathtt{Strat}$.*

Proof for Lemma 3. Follows directly from Definition 10 $\Diamond$

V. Non-interference for Components

After defining the concept of components, compositions and the cooperation a component expects from its environment, we want to develop a notion of non-interference for components under cooperative environments into consideration. We again use strategies to model the environment, but restrict the set of strategies to those who provide the guarantees assumed by a component.

We use cooperative strategies to define non-interference for a component conditional to cooperation.

Definition 11 (Cooperation-Non-Interference)**.** *A composition c is* non-interferent*, if $c_{LTS} \in \mathtt{COOP}_c$-NI.*

Since for compositions that do not require a service, every strategy is a cooperative strategy (Lemma 3), from cooperative non-interference, general non-interference follows.

Lemma 4. *For a component c with $\texttt{req}_c = \emptyset$ it holds that $c_{LTS} \in \texttt{COOP}_c$-NI $\Leftrightarrow c_{LTS} \in \texttt{Strat}$-NI*

Proof for Lemma 4. Follows from Definition 11 and Lemma 3. ◊

Cooperation-non-interference is compositional for components under synchronized parallel composition.

Theorem 3 (Composition Non-Interference)**.** *For a composition $d = p_A[\![s]\!]p_B$ with $p_A \in \texttt{COOP}_{p_A}$-NI, $p_B \in \texttt{COOP}_{p_B}$-NI and p_A, p_B* visibility-preserving *then $d \in \texttt{COOP}_d$-NI*

The proof for Theorem 3 can be found in the appendix.

VI. NI FOR SERVICES

Analyzing non-interference for entire components may be tedious, depending on the complexity of $\sim$. Especially, we do expect that complex definitions of $\sim$, although useful for specification, are hard to formalize. We expect this in particular for typical enforcement methods like type systems, especially, if high precision is necessary. It would therefore be beneficial, if we could utilize an additional dimension of modularity.

The next natural level of modularity are services. Services are rather simple programs in that they are deterministic and terminating. In this section, we provide a non-interference property for services, which allows composition of services to non-interferent components.

Non-interference properties for terminating, deterministic programs can be found in literature in many different shapes. Our notion presented here is inspired by non-interference for batch programs, which we extend by message passing. In this notion, a program, or service, is non-interferent, if it terminates in equivalent post-states after being started in equivalent pre-states.

Different methods for enforcement are provided, for example type systems, program dependency graphs and theorem proving approaches, differing in precision, specification overhead and manual interaction. We do not limit ourselves to a specific enforcement method, but restrict ourselves to the general idea.

The equivalence of states σ, σ' is in our setting defined by an equivalence relation $\approx$. While the definition of the partitioning of the state in a high and a low part is considered often as specified security property of a program, we merely consider this partition as a technical necessity. We do not put any focus on the intended meaning of a high and low partitioning here, as long as there exists some suitable equivalence relation on states.

We define two properties for services. First, we require a service not to reveal its execution, if the environment does not.

Definition 12 (Strictly Visibility-preserving Service)**.** *A service $serv$ is* visiblity-preserving *with respect to $\sim$ and $\approx$ if*

$$\begin{aligned}\forall \sigma, \sigma', t, t' \cdot \langle handler_{serv}; \sigma\rangle \xrightarrow{t \frown t'} \langle SKIP; \sigma'\rangle \implies \\ (t \rhd \mathbb{I} \sim \langle\rangle \implies t \rhd \mathbb{O} \sim \langle\rangle \ \wedge \\ t \frown t' \rhd \mathbb{I} \sim \langle\rangle \implies \sigma \approx \sigma')\end{aligned}$$

The first part of Definition 12 states that if all inputs provided by the environment to the service so far are invisible, the service may only provide invisible outputs. This condition is a more strict version of Definition 7. Additionally to the condition that a service called invisibly, terminates invisibly, Definition 12 requires all intermediate service calls to be invisible. In the context of cooperative strategies, a component can not be non-interferent, if a service makes a visible service call. Since we are about to define a non-interference definition for services, which is independent from strategies, we have to make this property explicit.

Additionally, if only invisible input were sent to a service, which terminates, the final state must be equivalent to the final state. This last condition is necessary in order to provide sequential compositionality and is formalized in the second part of Definition 12.

Second, we define non-interference for a service according to the classic definition of non-interference for batch programs and we add the consideration of equivalent traces.

Definition 13 (Service Non-Interference)**.** *A Service $serv$ is non-interferent with respect to $\sim$ and $\approx$, written $serv \in \texttt{SNI}^{\approx}_{\sim}$ iff it is* strictly visibility-preserving *with respect to $\sim$ and $\approx$ and*

$$\forall \sigma_1, \sigma_2, \sigma_1', \sigma_2', t_1, t_2 \cdot \sigma_1 \approx \sigma_2 \wedge \tag{7}$$
$$\langle handler_{serv}; \sigma_1\rangle \xrightarrow{t_1} \langle SKIP; \sigma_1'\rangle \ \wedge \tag{8}$$
$$\langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2} \langle SKIP; \sigma_2'\rangle \tag{9}$$
$$\implies \tag{10}$$
$$(t_1 \rhd \mathbb{I} \sim t_2 \rhd \mathbb{I} \implies \sigma_1' \approx \sigma_2') \ \wedge \tag{11}$$
$$(\forall t_1' \leq t_1, t_2' \leq t_2 \cdot \ t_1' \rhd \mathbb{I} \sim t_2' \rhd \mathbb{I} \implies \tag{12}$$
$$\exists t_1' \frown t_1'' \leq t_1, t_2' \frown t_2'' \leq t_2 \cdot \tag{13}$$
$$t_1' \frown t_1'' \sim t_2' \frown t_2'') \tag{14}$$

A service started in two equivalent states (7) has to terminate (9) in equivalent states, if the input provided by the environment is equivalent for both runs (11). Implicitly, this condition encodes a well-behaving environment in the sense that we assume the environment not to leak information.

The second condition ensures that the service does not leak information by providing non-equivalent output to the environment after receiving equivalent input. Lines 12 to 14 ensure that t_1 and t_2 are equivalent up to the first not equivalent input. For all prefixes of the two traces produced during execution which got provided equivalent input (12), the traces either are equivalent, or at least there are further events in the traces such that the traces can become equivalent (14). We

give in the condition the possibility that both prefixes can be extended. In fact, it is sufficient to only extend the prefix whose equivalence projection is shorter. But phrasing this formally does not result in a simpler formula.

Now, we want to characterize components, which only have non-interferent services with respect to the some equivalence relation $\approx$.

Definition 14 (Component State Non-interference)**.** *A component c is $(\sim, \approx)$-NI if $\forall serv \in \mathtt{prov}_c \cdot serv \in \mathtt{SNI}^{\approx}_{\sim}$.*

Given two equivalent cooperative strategies after observing equivalent traces, calling a service. Then, a component in $(\sim, \approx)$-NI produces equivalent traces under these environments and the services terminate after communicating equivalent traces.

Lemma 5. *Given $\omega_1 \sim \omega_2 \in \mathtt{COOP}$ with respective previous observations $p_1 \sim p_2$, $\sigma_1 \approx \sigma_2, \sigma_1'$, non-interferent service $serv$ and previous observations. Then*

$$\begin{aligned}
&\forall t_1 \cdot \langle handler_{serv}; \sigma_1 \rangle \xrightarrow{t_1} \langle SKIP; \sigma_1' \rangle \implies \\
&\quad (t_1 \sim \langle\rangle \wedge \sigma_1 \approx \sigma_1') \vee \\
&\quad \exists t_2, \sigma_2' \cdot \langle handler_{serv}; \sigma_2 \rangle \xrightarrow{t_2} \langle SKIP; \sigma_2' \rangle \wedge \\
&\qquad t_1 \sim t_2 \wedge \sigma_1' \sim \sigma_2'
\end{aligned}$$

Intuitively, the lemma holds for traces t_1 with an invisible initial message trivially, since the service has to produce an invisible trace according to Definition 12. In the case of the first message of t_1 being visible, we can show by induction for every prefix of t_1 that $\langle handler_{serv}; \sigma_2 \rangle$ accepts an equivalent trace t_2. The formal proof can be found in the Appendix.

We now can show that Definition 14 implies Cooperation-Non-Interference.

Theorem 4. *Given an equivalence relation $\approx$ such that c is $(\sim, \approx)$-NI, then $\langle c_{body}; \sigma_0 \rangle \in \mathtt{COOP}_c$-NI.*

The proof for Theorem 4 requires some additional technical lemmas and definitions. The interested reader may be referred to the Appendix.

Definition 13 can be seen as a proof obligation for services with respect to equivalence relations on messages and states of a component. Services in our setting are rather simple programs, since they are deterministic and terminating. Complex systems are created by combining services and components.

We gain according to Theorem 4 non-interference for a component by non-interferent services. So proving non-interference for an entire system is reduced to an analysis of single services, and a comparison of the state equivalence relation on states for services provided by a component. System non-interference then is for free by our compositionality results.

We do not fix the method for analysis of non-interference for services. This way, the best suitable method for a concrete service can be chosen. In some cases, the method of choice may be type systems or program dependency graphs. In cases of complicated security specifications, theorem proving approaches might be necessary.

VII. Related Work

Non-interference in general is a well-researched security property. The origins go back to *strong dependency* by Cohen [6] and the definition of non-interference by Goguen and Meseguer [7]. State-of-the-art research, as far as needed in this work, can be separated into (1) non-interference for interactive programs exchanging parametrized messages with its environment, where secrets are inputs and outputs during the run of a program; and (2) non-interference for batch-programs, where partly high and partly low input is provided as a state at the start of the program and the security property has to hold in the state after termination of the program.

Work on non-interference for programs with intermediate communication with its environment is manifold. Non-interference is discussed using event systems, for example, [8], [9], or process calculi, e.g., [10]–[12]. In both contexts, the environment is not modeled explicitly, but as traces or streams of input and output events.

The work closest to ours uses labeled transition systems as representations of programs and explicitly takes the environment of a program in the form of strategies into consideration. Modeling the environment by using strategies was pioneered by Wittbold et al. in [13]. Here, the environment is separated into high and low users of a system, each modeled as a strategy, and providing high and low input respectively. O'Neill et al. [14] present a formal analysis of non-interference for interactive programs in the presence of strategies. Clark et al. show in [5] that for deterministic programs, it is sufficient to model the environment as input-streams. Input-streams, in contrast to strategies, do not take the observation of an execution of the program into consideration, i.e., all input can be predetermined.

Rafnsson et al. add an additional dimension to the specification of parameters of messages as low and high type. In [3], they define the presence of messages as a possible secret, which leads to strategies, which are able to block program execution by not providing further input on a channel. The resulting non-interference property is very restrictive. Nearly all programs receiving intermediate secret messages as inputs followed by low outputs are insecure. Take, for example the program $\mathtt{read}(x \leftarrow \alpha); \mathtt{read}(y \leftarrow \beta); \mathtt{write}(1 \rightarrow \gamma)$, where an event on channel γ reveals that an event on channel β previously was provided to the program. If the presence of events on channel β is high, the observational communication has to be equivalent, independent from communication on this channel.

By using cooperative environments, we make the non-interference property more applicable in cases where we have some reason to assume a cooperative environment. More concretely, we employ contracts which are assumed by component-bases system engineering to hold for every environment the component is deployed in. While the call of a service is still considered a possible secret, the termination of a called service is not and therefore the program can not be

blocked by receiving the service termination as a high input. After composition of components, the fact that a service is called still is a secret and can not be observed by an attacker.

Further, we extend the work in [3] with declassification of information, according to Sabelfeld et al. [4] in the *what-dimension*. By using equivalence relations for the specification of secret information instead of previously used type systems. This extension allows us to specify parts of communicated parameters as high and low and preserve this specification over interface boundaries between components. Our extension is a generalization of the three dimension of high and low for communicated content and the secrecy of the existance of a message in the sense that we can express specifications according to the type system proposed by Rafnsson et al. with our equivalence relations. Nevertheless, the compositionality proof performed in [3] does not hold in our more general case, so we provide a new compositionality proof.

Vanhoef et al. [15] provide a similar notion of declassification which allows declassification of partial information using a *project* function for specification. Vanhoef et al. additionally allow the declassification of aggregated information over a history of events, making the declassification policy stateful. Their work builds on results by Sabelfeld and Sands [16], who allow the specification of information flow properties using partial equivalence relations for sequential batch programs. For enforcement of the policy, they propose a dynamic approach based on secure multi-execution, but do not provide a compositionality result for their non-interference property.

In contrast to Vanhoef et al. we aim for re-usability of components in different contexts, possibly with several different security lattices. In this setting a dynamic enforcement of non-interference using secure multi-execution is not practical, since we expect the cost of multi-execution to be too high in this case. We aim for a static and reusable analysis of components, which makes a compositional non-interference property necessary. Therefore, we prove compositionality for our extension of the strategy-based approach by Rafnsson et al. In contrast to Sabelfeld and Sands, we consider interactive programs instead of batch programs.

When considering compositionality of services, we extend non-interference for batch programs with intermediate message passing. A discussion on non-interference in batch programs can be found in [17]–[20]. Here two runs of a program started in two equivalent, but underspecified, initial states are compared. The program is secure, if the terminal states of both runs are equivalent with respect to some specification of secrets in the state. Sabelfeld and Sands [16] propose the PER model for information declassification using partial equivalence relations, but without interactive message passing with the environment. We extend this notion of non-interference for batch programs with intermediate event communication and we relate it to non-interference for interactive programs by providing a compositionality result for components, which states that non-interferent batch programs result in non-interferent interactive programs in the case of components.

VIII. Conclusion

We have presented a framework to express non-interference for interactive programs, allowing what-declassification of information provided as input messages. Further, we have applied this framework to *components*, where components are defined as a sequential composition of sequential, terminating, and deterministic programs. The component model is inspired by the Java Enterprise Edition, a common framework for implementing distributed enterprise systems, And assumptions made on the environment of a component are inspired by Szyperski-like components. In Theorem 3 we have shown compositionality of non-interference in the presence of a cooperative environment. To analyze non-interference for components, we provide a non-interference property for services and in Theorem 4 we show compositionality of non-interference of services with shared states within a component.

Future work will be concerned with different topics. For one, requiring components to provide their services sequentially is a limitation in a more general case. We would like to develop non-interference properties for services executed in parallel within one component, while sharing a common state. A modification of the *rely-guarantee* approach [21]–[23] seems promising as a tool to limit program analysis to single services plus a compositionality condition. Parallel execution of services would also allow us to lift the restriction of non-reentrant components.

Further, Theorem 4 requires services within one component to be non-interferent with respect to one and the same equivalence relation on states. We expect it to be tedious, especially when using theorem prover approaches for enforcement, to find an equivalence relation which is suitable for all services within one component. We assume that a service is in general non-interferent with respect to an entire set of equivalence relations, which can be described in an abstract way. We will analyze how we can make use of this observation in order to show non-interference of services with respect to a set of equivalence relations, while showing compositionality of services by checking simpler conditions on compatibility of the sets of equivalence relations of different services.

And finally, we plan to develop and implement analysis techniques for of services (cf. Definition 13) on code level. Type systems are not precise enough for real world programs, since semantic declassification as defined in this paper is hard to cover with type systems. Instead, there is promising work using theorem provers to analyze non-interference of programs with what-declassification (e.g., [24], [25]). These theorem provers support a rich subset of Java and therefore can analyze programs implemented in real world programming languages. Since the theorem prover approaches do not support reasoning about programs with message passing, it is crucial to lift this limitation.

References

[1] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-oriented Programming*. Pearson Education, 2002.

[2] EJB 3.1 Expert Group, *JSR 318: Enterprise JavaBeans, Version 3.1*, Sun Microsystems, 2009. [Online]. Available: https://jcp.org/aboutJava/communityprocess/final/jsr318/

[3] W. Rafnsson, D. Hedin, and A. Sabelfeld, "Securing interactive programs," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, 2012, pp. 293–307.

[4] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *J. Comput. Secur.*, vol. 17, no. 5, pp. 517–548, Oct. 2009.

[5] D. Clark and S. Hunt, "Non-interference for deterministic interactive programs," in *Formal Aspects in Security and Trust*, ser. Lecture Notes in Computer Science, P. Degano, J. Guttman, and F. Martinelli, Eds. Springer Berlin Heidelberg, 2009, vol. 5491, pp. 50–66.

[6] E. Cohen, "Information transmission in computational systems," *SIGOPS Oper. Syst. Rev.*, vol. 11, no. 5, pp. 133–139, Nov. 1977.

[7] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[8] H. Mantel, "Possibilistic definitions of security – an assembly kit," in *13th IEEE Computer Security Foundations Workshop (CSFW '00)*, 2000, pp. 185–199.

[9] A. Sabelfeld and H. Mantel, "Static Confidentiality Enforcement for Distributed Programs," in *Static Analysis*, ser. Lecture Notes in Computer Science, M. Hermenegildo and G. Puebla, Eds. Springer Berlin Heidelberg, 2002, vol. 2477, pp. 376–394.

[10] R. Focardi and R. Gorrieri, "A classification of security properties for process algebras," *Journal of Computer Security*, vol. 3, pp. 5–33, 1994.

[11] P. Y. A. Ryan and S. A. Schneider, "Process algebra and non-interference." in *CSFW*. IEEE Computer Society, 1999, pp. 214–227.

[12] F. Pottier, "A simple view of type-secure information flow in the pi-calculus," in *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, ser. CSFW '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 320–.

[13] J. T. Wittbold and D. M. Johnson, "Information flow in nondeterministic systems," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1990, pp. 144–161.

[14] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. Piscataway, NJ, USA: IEEE Press, Jul. 2006, pp. 190–201.

[15] M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk, "Stateful declassification policies for event-driven programs," in *2014 IEEE 27th Computer Security Foundations Symposium (CSF 2014)*. IEEE, July 2014, pp. 293–307.

[16] A. Sabelfeld and D. Sands, "A PER model of secure information flow in sequential programs," *Higher-Order and Symbolic Computation*, vol. 14, no. 1, pp. 59–91, 2001.

[17] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, 2004, pp. 100–114.

[18] R. Joshi and K. R. M. Leino, "A semantic approach to secure information flow," *Sci. Comput. Program.*, vol. 37, no. 1-3, pp. 113–138, 2000.

[19] T. Amtoft and A. Banerjee, "Information flow analysis in logical form," in *Static Analysis*, ser. Lecture Notes in Computer Science, R. Giacobazzi, Ed. Springer Berlin Heidelberg, 2004, vol. 3148, pp. 100–115.

[20] Á. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*, 2005, pp. 193–209.

[21] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, pp. 596–619, Oct. 1983.

[22] C. Stirling, "A generalization of Owicki-Gries's Hoare logic for a concurrent while language," *Theoretical Computer Science*, vol. 58, no. 1–3, pp. 347 – 359, 1988.

[23] K. Stølen, "A method for the development of totally correct shared-state parallel programs," in *CONCUR '91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991, Proceedings*, 1991, pp. 510–525.

[24] C. Scheben and P. H. Schmitt, "Verification of information flow properties of Java programs without approximations," in *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*, 2011, pp. 232–249.

[25] D. Grahl, "Deductive verification of concurrent programs and its application to secure information flow for Java," Ph.D. dissertation, Karlsruhe Institute of Technology, 29 Oct. 2015.

APPENDIX

Before we provide the proofs, we introduce some notions and show some supporting lemmas.

A. Preliminaries

The following lemmas allow some shortcuts in the arguments in the proofs to come.

Lemma 6. *Given $\omega, \omega' \in \texttt{Strat}$. Then, the following notions are equivalent:*

1) $\omega \sim \omega'$

2) $\forall m \nsim \Box, t \cdot \omega(t) \rhd [m] = \emptyset \Leftrightarrow \omega'(t) \rhd [m] = \emptyset$

3) $\forall t, t' \cdot t \sim t' \implies \omega(t) \sim \omega'(t')$

Proof for Lemma 6. $1 \Leftrightarrow 2$:
Assume $\omega \sim \omega'$. Then, by Definition 3, $\forall t \cdot \omega(t) \sim \omega'(t)$, which is by definition of low-projection $\forall t \cdot \{[m] \mid \exists n \sim m \in \omega(t) \wedge m \notin [\Box]\} = \{[m] \mid \exists n \sim m \in \omega'(t) \wedge m \notin [\Box]\}$, which is equal to $\forall t, [m] \neq [\Box] \cdot \omega(t) \rhd [m] = \emptyset \Leftrightarrow \omega'(t) \rhd [m]$ which is equivalent to $\forall [m], t \cdot [m] \neq [\Box] \Leftrightarrow \omega(t) \rhd [m] \sim \omega'(t) \rhd [m]$

$1 \implies 3$:
$\omega \sim \omega' \wedge t \sim t' \implies \omega(t) \sim \omega(t') \wedge \omega'(t) \sim \omega'(t')$ Since ω and ω' are strategies. $\omega \sim \omega'$, therefore $\omega(t) \sim \omega'(t)$. And by transititvity: $\omega(t) \sim \omega'(t) \sim \omega'(t')$
$3 \implies 1$:
$\forall t \sim t' \cdot \omega(t) \sim \omega'(t')$, so especially $\forall t \cdot \omega(t) \sim \omega'(t)$ and by definition of $\sim$ on sets of messages: $\forall t \cdot \omega(t)\restriction_\ell = \omega'(t)\restriction_\ell$ which is the definition of $\sim$ on strategies. $\lozenge$

We say a trace t contains a message m, written $m \in t$, iff $\exists t', t'' \cdot t' \frown m \frown t'' = t$.

We say a message $m \in \mathbb{M}$ is *visible*, if $m \nsim \Box$. The term $t \rhd \blacksquare$ yields a trace without invisible messages in t. The formal definition is

$$\langle\rangle \rhd \blacksquare := \langle\rangle$$
$$(m \frown t) \rhd \blacksquare := \begin{cases} t \rhd \blacksquare \text{ if } m \sim \Box \\ m \frown (t \rhd \blacksquare) \text{otherwise} \end{cases}$$

If two traces are equivalent, then the amount of visible messages in both traces is equal.

Lemma 7. $\forall t \sim t' \cdot |t \rhd \blacksquare| = |t' \rhd \blacksquare|$

Proof. Follows from definition of equivalence of traces and the definition of the filter operation for visible messages. $\lozenge$

We frequently argue about *interleavings* in the following proofs. So, $t_1 \mid\mid\mid_t t_2$, iff t is an interleaving of t_1 and t_2.

As a shortcut, we define *prefix equivalence*, written $s \leq_\sim t$ as $\exists t_1 \leq t \cdot s \sim t_1$.

Also, an LTS is input neutral for equivalent and visible messages.

Lemma 8. *Given LTS p and $m, m' \in \mathbb{I}$ with $m \nsim \Box$ and $m' \sim m$. Then $p \xrightarrow{m} \implies p \xrightarrow{m'}$.*

Proof for Lemma 8. Follows directly from definition of LTS and restrictions on $\sim$. $\Diamond$

Non-interference of an LTS does not depend on feeding of invisible messages. If we can find an attack on an LTS, then there also exists an attack with a strategy that does not provide any secret input.

Lemma 9. *If (ω_1, ω_2, t) is an attack on p, then, for some ω_2' with $\forall t \cdot \omega_2'(t) = \omega_2(t) \setminus [\Box]$, (ω_1, ω_2', t) is an attack on p.*

Proof for Lemma 9. $\forall t' \cdot \omega_2' \models t' \implies \omega_2 \models t$. Therefore, if there is no trace with $\omega_2 \models p \xrightarrow{t}$, no t can exist with $\omega_2' \models p \xrightarrow{t}$. $\Diamond$

B. Proofs omitted in the main paper

Proof for Theorem 2. We proof $p_A \parallel p_B \notin \texttt{Strat}\text{-NI} \implies p_A, p_B \notin \texttt{Strat}\text{-NI}$.

Then there exists an attack (ω_1, ω_2, t) on $p_A \parallel p_B$. Especially, we know by Lemma 9 that ω_2 does not produce invisible input.

Assume towards contradiction $p_A, p_B \in \texttt{Strat}\text{-NI}$. Then by definition we have for $k \in \{A, B\}$: $\forall \omega_{1k}, \omega_{2k} \in \texttt{Strat} \cdot \omega_{1k} \sim \omega_{2k} \implies \forall t_{1k} \cdot \omega_{1k} \models p_k \xrightarrow{t_{1k}} \implies \exists t_{2k} \cdot \omega_{2k} \models p_k \xrightarrow{t_{2k}} \wedge t_{1k} \sim t_{2_k}$

Select t_{1A}, t_{1B} such that t is an interleaving of t_{1A} and t_{1B} and $p_A \xrightarrow{t_{1A}}$ and $s_B \xrightarrow{t_{1B}}$.

We now construct strategies $\omega_{1A}, \omega_{2A}, \omega_{1B}, \omega_{2b}$ such that $\omega_{1A} \sim \omega_{2A}$ and $\omega_{1B} \sim \omega_{2B}$.

Let $j \in \{1, 2\}, k, k' \in \{A, B\}, k \neq k'$.

$$\begin{aligned}\omega_{jk}(t) := \{m \mid \exists t_1' t_k', t_{k'}' \cdot t \sim t_k' \wedge t_k' \mid\mid\mid_{t_1'} t_{k'}' \\ \wedge\, t_k' \frown m \leq_\sim t_{1k} \wedge p_k \xrightarrow{t_k' \frown m} \\ \wedge\, t_{k'}' \leq_\sim t_{1k'} \wedge p_{k'} \xrightarrow{t_{k'}'} \\ \wedge\, t_1' \frown m \leq_\sim t_1 \wedge \omega_j \models p_A \parallel p_B \xrightarrow{t_1' \frown m}\}\end{aligned}$$

We have to show $\omega_{jk} \in \texttt{Strat}$, $\omega_{1k} \sim \omega_{2k}$, $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$ and $\omega_{1B} \models p_B \xrightarrow{t_{1B}}$ bevor we show that this contradicts $p_A, p_B \in \texttt{Strat}$.

Proof for $\omega_{jk} \in \texttt{Strat}$:
Let $t \sim t'$. Assume $m \in \omega_{jk}(t)$. Let $t_1', t_k', t_{k'}'$ be the witnesses for m above. Since $t \sim t'$ and $t \sim t_k'$ also $t' \sim t_k'$. Therefore $t_1', t_k', t_{k'}'$ is a witness for t' and $m \in \omega_{jk}(t')$ and $\omega_{jk}(t) \sim \omega_{jk}(t')$.

Proof for $\omega_{1k} \sim \omega_{2k}$:
Let t be arbitrary. Assume w.l.g. $m \in \omega_{1k}(t)$ and $(m \nsim \Box)$.

Let $t_1', t_k' t_{k'}'$ be the witness for m in the definition of ω_{1k}. Since $\omega_j \models p_A \parallel p_B \xrightarrow{t_1' \frown m}$ and $\omega_1 \sim \omega_2$ there exists an $m' \in \omega_2(t_1')$ with $m' \sim m$. By input neutrality, $p_A \parallel p_B \xrightarrow{t_1' \frown m'}$. Since $t_k' \frown m \sim t_k' \frown m'$, also $t_k' \frown m' \leq_\sim t_{1k}$. Similar since $t_1' \frown m \sim t_1' \frown m'$, also $t_1' \frown m' \leq_\sim t_1$. Lines 1 and 3 from the definition are independent from parameter j, therefore $m' \in \omega_{2k}(t)$.

Proof for $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$:
We show that $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$ We already have $p_A \xrightarrow{t_{1A}}, p_B \xrightarrow{t_{1B}}, t_{1A} \mid\mid\mid_{t_1} t_{1B}$ and $\omega_1 \models p_A \parallel p_B \xrightarrow{t_1}$.

Induction over $n = |t_{1A} \rhd \mathbb{I}|$
$n = 0$: Since $p_A \xrightarrow{t_{1A}}$ and t_{1A} has no inputs, trivially it holds $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$. $n + 1$, given n: Assume $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$ with $|t_{1A}' \rhd \mathbb{I}| = n$. For some t_{1A}'' with $|t_{1A}'' \rhd \mathbb{I}| = 0$ and some $m \in \mathbb{I}$ we get $t_{1A} = t_{1A}' \frown m \frown t_{1A}''$.

For some $t_{1B}' \leq t_{1B}$ and $t_1' \leq t_1$ we have $t_{1A}' \mid\mid\mid_{t_1'} t_{2B}'$ that $\omega_{1A} \models p_A \parallel p_B \xrightarrow{t_1' \frown m}$. By $p_A \xrightarrow{t_{1A}}$ and $p_B \xrightarrow{t_{1B}}$, we get $p_A \xrightarrow{t_{1A}' \frown m}$ Since $u \leq u' \implies u \leq_\sim u'$, we get by definition of ω_{1A}: $m \in \omega_{1A}(t_{1A}')$. Therefore $\omega_{1A} \models p_A \xrightarrow{t_{1A}' \frown m}$. And since t_{1A}'' does not have inputs, $\omega_{1A} \models p_A \xrightarrow{t_{1A}' \frown m \frown t_{1A}''}$.

The proof for $\omega_{1B} \models p_B \xrightarrow{t_{1B}}$ can be obtained by swapping A and B.

Now, we can show that $p_A \notin \texttt{Strat}\text{-NI}$ or $p_B \notin \texttt{Strat}\text{-NI}$

We have assumed towards contradiction that $p_A, p_B \in \texttt{Strat}\text{-NI}$. Since $\omega_{1k} \sim \omega_{2k}$, there exist $t_{1k} \sim t_{2k}$ such that $\omega_{2k} \models p_k \xrightarrow{t_{2k}}$. We now show that there exists $t_2 \sim t_1$ with $\omega_2 \models p_A \parallel s_B \xrightarrow{t_2}$ which contradicts the original assumption.

We assume $|t_{2k} \rhd \mathbb{I}| > 0$. Let $t_{2k} = t_{2k}' \frown m_k \frown t_{2k}''$ with $|t_{2k}'' \rhd \mathbb{I}| = 0$ and $m_k \in \mathbb{I}$. By definition of ω_{2k}, we have $m_A \in \omega_{2A}(t_{2A}')$ and $m_B \in \omega_{2B}(t_{2B}')$. Therefore, there exist t_1^A, t_1^B such that $t_1^A \frown m_A \leq_\sim t_1$ and $t_1^B \frown m_B \leq_\sim t_1$ and $\omega_2 \models p_A \parallel p_B \xrightarrow{t_1^A \frown m_A}$ and $\omega_2 \models p_A \parallel p_B \xrightarrow{t_1^B \frown m_B}$. Since $m \in \omega_2(t) \implies m \nsim \Box$, we know ${t_1}^A \frown m_A \rhd \mathbb{I} \rhd \blacksquare = {t_1}^A \frown m_A \rhd \mathbb{I}$ and $t_1^B \frown m_B \rhd \mathbb{I} \rhd \blacksquare = t_1^B \frown m_B \rhd \mathbb{I}$

By definition, we either get $t_1^A \frown m_A \leq_\sim t_1^B \frown m_B \leq_\sim t_1$ or $t_1^B \frown m_B \leq_\sim t_1^A \frown m_A \leq_\sim t_1$. W.l.g. $t_1^B \frown m_B \leq_\sim t_1^A \frown m_A \leq_\sim t_1$. Therefore, we get $t_1^A \frown m_A \rhd \mathbb{I} \sim t_1 \rhd \mathbb{I}$. Thus, there also exists t_1', t_1'' such that $t_1' \frown t_1'' = t_1$ and $t_1'' \rhd \mathbb{I} \sim \langle\rangle$ and $t_1' \sim t_1^A \frown m_A$. Now there is some u_1'' with $u_1'' \sim t_1''$ and $t_{2A}'' \mid\mid\mid_{u_1''} t_{2B}''$. Since $|u_1'' \rhd \mathbb{I}| = 0$ and $\omega_2 \models (p_A \parallel p_B) \xrightarrow{t_1^A \frown m_A}$, we also get $\omega_2 \models (p_A \parallel p_B) \xrightarrow{t_1^A \frown m_A \frown u_1''}$. But $t_1^A \frown m_A \frown u_1'' \sim t_1$, which contradicts the original assumption in this proof. Thus, either $p_A \notin \texttt{Strat}\text{-NI}$ or $p_B \notin \texttt{Strat}\text{-NI}$ $\Diamond$

Proof for Theorem 3. We proof the contrapositive. Let $d \notin \texttt{COOP}_d\text{-NI}$. Therefore, it exists an attack on d: $(\omega_1, \omega_2, t_1)$ with $\omega_1, \omega_2 \in \texttt{COOP}_d$, $\omega_1 \models d_{LTS} \xrightarrow{t_1}$ and $\forall t_2 \cdot \omega_2 \models d_{LTS} \xrightarrow{t_2} \implies (t_2 \nsim t_1)$.

Then, there exist traces t_{1A}, t_{1B} such that $(t_{1A} [\![s]\!] t_{1B}) = t_1$ and $p_{A_{LTS}} \xrightarrow{t_{1A}}$ and $p_{B_{LTS}} \xrightarrow{t_{1B}}$.

As in proof for Theorem 2, we construct strategies which then result in attacks on p_A or p_B.

First we construct strategies for p_A and p_B:

Let $j \in \{1, 2\}$, $k, k' \in \{A, B\}$, $k \neq k'$. We define strategies

ω_{jk}:

$$\omega'_{jk}(t) := \{m \mid \exists t'_1, t'_k, t'_{k'} \cdot t \sim t'_k \wedge t'_k[\![s]\!]t'_{k'} = t'_1 \\ \wedge\, t'_k \frown m \leq_\sim t_{1k} \wedge s_k \xrightarrow{t'_k \frown m} \\ \wedge\, t'_{k'} \leq_\sim t_{1k'} \wedge s_{k'} \xrightarrow{t'_{k'}} \\ \wedge\, t'_1 \frown m \leq_\sim t_1 \wedge \omega_j \models p_A \parallel p_B \xrightarrow{t'_1 \frown m}\}$$

We extend the strategies ω_{jk} with the terminating events needed to make the strategies cooperative.

$$\omega_{jk}(t) := \omega'_{jk}(t) \cup \\ \{m \mid \exists t', t'', v, w, serv \cdot t = t' \frown Ini(serv)!v \frown t' \wedge \\ (Fin(serv)?w' \notin t') \wedge Ini(serv)!v \sim \Box \wedge \\ m = Fin(serv)?w \wedge m \sim \Box \wedge p_k \xrightarrow{t \frown m} \wedge \\ \text{not}(\exists u, w'' \cdot (Fin(serv)?w' \notin u) \wedge \\ Fin(serv)?w' \in \omega'_{jk}(t \frown u) \wedge \omega'_{jk}(t) \models p_k \xrightarrow{t \frown u} \wedge)\}$$

We have to show $\omega_{jk} \in \texttt{Strat}$, $\omega_{jk} \in \texttt{COOP}_{p_k}$ and $\omega_{1k} \sim \omega_{2k}$.

Proof for $\omega_{jk} \in \texttt{Strat}$:
See proof of Theorem 2.

Proof for $\omega_{jk} \in \texttt{COOP}_{p_k}$:
$\omega_{jk} \in \texttt{Strat}$, so it is left to show two properties from Definition 10
Definition 10, Lines 2, 3):
We have to show

$$\forall t, t', serv, \sigma, v \cdot serv \in \texttt{req}_c \wedge \\ \omega_{jk} \models p_k \xrightarrow{t \frown Ini(serv)!v \frown t'} \wedge Fin(serv) \notin t' \\ \implies \exists t'' \cdot \omega_{jk} \models p_k \xrightarrow{t \frown Ini(serv)!v \frown t' \frown t''} \wedge \\ Fin(serv)?v' \in \omega_{jk}(t \frown Ini(serv)!v \frown t' \frown t'')$$

Select $t, t', serv, v$ such that $serv \in \texttt{req}_{p_k}$, $\omega_{jk} \models p_k \xrightarrow{t \frown Ini(serv)!v \frown t'}$, $Fin(serv) \notin t'$.

Case 1: $k = A$ Since $\omega_{jA} \models s_A \xrightarrow{t \frown Ini(serv)!v \frown t'}$, there exists witnesses $t'_1, t'_k, t'_{A'}$, such that $t \frown Ini(serv)!v \frown t' \sim t'_A$. Therefore, it exists $t'_A = u \frown Ini(serv)!v' \frown u'$ such that $t \sim u, Ini(serv)!v \sim Ini(serv)!v', t' \sim u'$. Therefore, u' can contain $Fin(serv)!w$ only if $Fin(serv)!w \sim \Box$.

Case 1.1: $(Ini(serv)!v \nsim \Box)$:
Since $\omega_j \in \texttt{COOP}_d$ and $(Ini(serv)!v \nsim \Box)$ (Second condition of Definition 10), $Fin(serv)!w$ is not in u'. Again, since $\omega_j \in \texttt{COOP}_d$, there exists u'', such that $\omega_j \models d \xrightarrow{u \frown Ini(serv)!w \frown u' \frown u''} \wedge Fin(serv)?w' \in \omega_j(u \frown Ini(serv)!w \frown u' \frown u'')$. Also, due to $\omega_j \in \texttt{COOP}_d$ and $(Ini(serv)!w \nsim \Box)$, also $(Fin(serv)?w' \nsim \Box)$. Similar to above, we can again split this trace in u_A, u_B such that it is accepted by p_A. Since $u \sim t, Ini(serv)!v \sim Ini(serv)!v', t' \sim u'$, $t'_1, t'_k, t'_{A'}$ is also a witness for $Fin(serv)!w' \in \omega_{jA}$

Case 1.2: $Ini(serv)!v \sim \Box$
Due to the extension of ω'_{jk} to ω_{jk} as constructed above, there is a trace u such that the terminating message can be consumed.

Case 2: $k = B$
Proof is similar to Casees 1.1 and 1.2.

Definition 10, lines 4, 6):

$$\forall t, t', serv, \sigma \cdot serv \in \texttt{req}_c \wedge \omega_{jk} \models p_k \xrightarrow{t \frown Ini(serv)!v \frown t'} \wedge \\ Fin(serv) \notin t' \wedge Fin(?)v' \in \omega_{jk}(t \frown Ini(serv)!v \frown t') \\ \implies (Ini(serv)!v \sim \Box \Leftrightarrow Fin(serv)?v' \sim \Box)$$

This follows by construction of ω_{jk}. ω'_{jk} is constructed from ω_j, it is ensured that visible calls are terminated visible. Since ω'_{jk} is only extended with invisible termination events, if the original call was also invisible, the claim holds.

Proof for $\omega_{1k} \sim \omega_{2k}$:
See proof of Theorem 2. The definition of ω'_{jk} is basically the same. Note that the extension to ω_{jk} only adds invisible events, therefore $\omega'_{jk} \sim \omega_{jk}$.

Form the four ω_{jk},there results an attack p_A or p_B similair to the proof for Theorem 2. ◊

Proof for Lemma 5. Let t_1 with first message m be arbitrary.

Case 1: $m \sim \Box$. Since $serv$ is non-interferent, $serv$ is also visibility-preserving. By definition Definition 12 $t_1 \sim \langle\rangle$ and $\sigma_1 \sim \sigma'_1$.

Case 2: $m \nsim \Box$.

For all traces $t'_1 \leq t_1$ exists a trace t_2 such that $t'_1 \sim t'_2$ and $\omega_2\langle handler_{serv}; \sigma_2\rangle \xrightarrow{t'_2}$. Induction over $n = |t'_1|$.

Start: $n = 1$. Since m is visible, $\omega_1 \sim \omega_2$ and $m \in \omega_1(p_1)$ and $p_1 \sim p_2$, there exists $m_2 \in \omega_2(p_2)$ with $m \sim m_2$. Therefore $t'_2 = m_2$.

Step: $n + 1$ $t'_2 \sim t'_1$. $t'_1 \frown m \leq t_1$. Case 2.1: $m \sim \Box$. t'_2 is the witness.

Case 2.2: $m \nsim \Box$.

Case 2.2a: $m \in \mathbb{I}$. Therefore $t'_1 = t''_1 \frown o_1$, $o_1 \in \mathbb{O}$ Since $\omega_1 \in \texttt{COOP}$, $(o_1 \nsim \Box)$. Since $t'_1 \sim t'_2$ and $\omega_2 \in \texttt{COOP}$, $t'_2 = t''_2 \frown o_2$, $(o_2 \nsim \Box)$. Again, since $m \in \omega_1(p_1 \frown t'_1)$ and $\omega_1 \sim \omega_2$, there exists $m' \in \omega_2(p_2 \frown t'_2)$ with $m' \sim m$. Therefore $t'_1 \frown m \sim t'_2 \frown m'$ and $\omega_2 \models \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t'_2 \frown m'}$.

Case 2.2b: $m \in \mathbb{O}$. $t'_1 \sim t'_2$, thus $t'_1 \frown m \rhd \mathbb{I} \sim t'_2 \rhd \mathbb{I}$. Since $serv$ is non-interferent and $\sigma_1 \approx \sigma_2$, there exists t''_2, m' with $t'_2 \frown t''_2 \leq t_2$ with $t'_1 \frown m \sim t'_2 \frown m' \frown t''_2$ and $\langle handler_{serv}; \sigma_2\rangle \xrightarrow{t'_2 \frown m'}$. If m' visible, $m' \sim m$ and $m_2 \in \mathbb{O}$ and therefore $\omega_2 \models \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t'_2 \frown m'}$. If $m' \sim \Box$ and $m' \in \mathbb{O}$, then, $\omega_2 \in \texttt{COOP}$ there exists $m'' \sim \Box$ with $m'' \in \omega_2(p_2 \frown t'_2 \frown m' \frown m'')$ and therefore $\omega_2 \models \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t'_2 \frown m' \frown m''}$. It holds $t'_1 \frown m \rhd \mathbb{I} \sim t'_2 \frown m' \frown m'' \rhd \mathbb{I}$. Since $serv$ is terminating, $t'_2 := t'_2 \frown m' \frown m''$. Recursively case 2.2b ensures m' is visible. Therefore $t'_1 \frown m \sim t'_2 \frown m'$ and $\omega_2 \models \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t'_2 \frown m'}$

Since t_1 terminates on a visible output (Definition 7) and $t_2 \sim t_1$, t_2 is a terminating trace. Since $serv$ is non-interferent $\sigma'_1 \sim \sigma'_2$ ◊

Proof for Theorem 4. Instead, we show the contrapositive, i.e. we assume an attack to exist and show that this leads to a contradiction to $c \in (\sim, \approx)$-NI.

Given a component c and a $\texttt{COOP}_c - Attack$ (ω_1, ω_2, t) such that $\omega_1, \omega_2 \in \texttt{COOP}_c$, $\omega_1 \sim \omega_2$, $\omega_1 \models \langle c_{body}; \sigma_0 \rangle \xrightarrow{t}$ and $\forall t' \cdot \omega_2 \models \langle c_{body}; \sigma_0 \rangle \xrightarrow{t'} \implies (t \nsim t')$.

Select $t_1 \leq t$ the longest prefix of t for which an equivalent trace for ω_2 exists. Among all possible candidates for ω_2, select the one, which is equivalent to t_1 and has the most visible events in the trace. And finally, among those, we select the longest possible trace, meaning, there are no invisible events following t_2 for c under ω_2.

Formally:

1) $t_1 \leq t \wedge \forall t' \leq t \cdot \exists t'' \sim t' \cdot \omega_2 \models \langle c_{body}; \sigma_0 \rangle \xrightarrow{t''} \implies |t'| \leq |t_1|$.
2) $t_2 \sim t_1 \wedge$
3) $\omega_2 \models \langle c_{body}; \sigma_0 \rangle \xrightarrow{t_2}$
4) $\forall t' \leq_\sim t \cdot \omega_2 \models \langle c_{body}; \sigma_0 \rangle \xrightarrow{t'} \implies |t' \rhd \blacksquare| \leq |t_2 \rhd \blacksquare|$
5) $\forall t_2 \leq t' \wedge t' \sim t_1 \cdot \omega_2 \models \langle c_{body}; \sigma_0 \rangle \xrightarrow{t'} \implies t' = t_2$

We split t_i, $i \in \{1, 2\}$ in t_{ia} and t_{ib} such that t_{ia} finishes with a termiantion of a provided service and $t_i b$ does not terminate any service provided by c.

$t_i =: t_{ia} \frown t_{ib}, i \in \{1, 2\}$ such that $\langle c_{body}; \sigma_0 \rangle \xrightarrow{t_{ia}} \langle c_{body}; \sigma_i \rangle$ and $\langle serv_{i_{body}}; \sigma_i \rangle \xrightarrow{t_{ib}} \langle rest_i; \sigma_i' \rangle$.

This means, that there is a next event m in t and some event m' which might be consumed by c and provided by ω_2. I.e. let m, m' such that $t_1 \frown m \leq t$ and $\langle rest_2; \sigma_2' \rangle \xrightarrow{m'}$.

The event m' can not be invisible, because otherwise there would have existed a longer trace t_2, such that t_2 satisfies the conditions above.

We make a case distinction over $m \in \mathbb{I}$ and $m \in \mathbb{O}$ and show that both cases result in a contradiction to the original assumption.

Case 1: $m \in \mathbb{I}$, m' is visible: Since $\omega_1 \sim \omega_2$ and m visible, we know that there exists an $m' \in \omega_2(t_2)$ such that $m' \sim m$. Since $\omega_2 \in \texttt{COOP}_c$, we know that ω_2 can not block execution on visible input, bacause, if m is a service call, then m' is a call to the same service and otherwise m and m' are termination events on required services, which must not be blocked by service strategies.

So there exists a trace t_2' such that $t_2' \sim t_1$ and $\omega_2 \models \langle c_{body}; \sigma_0 \rangle \xrightarrow{t_2'}$, i.e. $\omega_2 \models \langle c_{body}; \sigma_0 \rangle \xrightarrow{t_2' \frown m'}$. But this means due to Lemma 7, that $|{t_2}' \frown m' \rhd \blacksquare| > |t_2 \rhd \blacksquare|$, which contradicts the construction of t_2.

Case 2: $m \in \mathbb{O}$: First we show, that $m' \in \mathbb{O}$. Since $serv_1$ is terminating, there exists a trace $t_{1a} \leq t_1'$ such that $\langle serv_{1_{body}}; \sigma_1 \rangle \xrightarrow{t_1'} \langle SKIP; \sigma_1'' \rangle$. Since $serv_1 \in \texttt{SNI}_\sim^{\approx}$ and, by induction, $\sigma_1 \approx \sigma_2$, we know that for all traces t_2' such that $\langle serv_{2_{body}}; \sigma_2 \rangle \xrightarrow{t_2'} \langle SKIP; \sigma_2'' \rangle$ it holds that ${t_1}' \rhd \mathbb{I} \sim {t_2}' \rhd \mathbb{I} \implies t_1' \sim t_2'$

$serv_2$ can not be blocked due to missing termination of a called service. Therefore, if $m' \in \mathbb{I}$, $m \in Ini(\texttt{prov}_c)$. Since $m \in \mathbb{O}$, $(t_{1b} \neq \langle\rangle)$, but $t_{1b} \sim \langle\rangle$, especially the initial message is invisible. This contradicts Definition 12.

Therefore $m' \in \mathbb{O}$ and by definition $\omega_2 \models \langle c_body; \sigma_0 \rangle \xrightarrow{t_2 \frown m'}$. Due to the construction of t_2, m' is visible.

According to Definition 8, there exists a trace t_2', such that ${t_2}' \rhd \mathbb{I} \sim {t_1}' \rhd \mathbb{I}$

Therefore, $t_2' \sim t_1'$, and it has to hold that $m' \sim m$, therefore $t_1 \frown m \sim t_2 \frown m'$, which is a contradiction to the construction of t_2.

$\Diamond$