SPaCIoS

Secure Provision and Consumption in the Internet of Services

FP7-ICT-2009-5, ICT-2009.1.4 (Trustworthy ICT)

Project No. 257876

www.spacios.eu

Deliverable D3.1
Test Case Generator

Abstract

This deliverable describes the automated reasoning techniques for test case generation developed within the SPaCIoS project. In particular, we present the approaches developed in SPaCIoS for both property-driven and vulnerability-drive test case generation. Moreover, we identify some directions of interest for future research and development.

Deliverable details

Deliverable version: *v1.0*
Classification: *public*
Date of delivery: *03.10.2013*
Due on: *30.09.2013*
Editors: *all*
Total pages: *49*

Project details

Start date: *October 01, 2010*
Duration: *36 months*
Project Coordinator: *Luca Viganò*
Partners: UNIVR, ETH Zurich, INP, KIT/TUM, UNIGE, SAP, Siemens, IeAT

SEVENTH FRAMEWORK PROGRAMME

(this page intentionally left blank)

Contents

List of Figures

List of Acronyms

AAT Abstract Attack Trace . . . 3
ASLan low level AVANTSSAR Specification Language . . . 3
ASLan(++) low(high) level AVANTSSAR Specification Language . . . 29
DSFlaw Data Sanitization Flaw . . . 31
FA Fact Assertion . . . 31
LTL Linear Temporal Logic . . . 25
SUT System Under Test . . . 29
XML eXtensible Markup Language . . . 30
XSLT Extensible Stylesheet Language Transformation . . . 30

1 Introduction

In this deliverable, we present the techniques and technologies involved in the advanced, automatic generation of test cases in the SPaCIoS project. Since test case generation is a core activity in SPaCIoS, we carried out considerable work to improve the existing approaches and, in the third project year, we investigated the different automated test case generation techniques considered in the project, along with possible extensions.

In Section 2, we describe the work done on property-driven test case generation. More specifically, Section 2.1 describes the research carried out on LTL separation for the manipulation of LTL goals, whereas in Section 2.2 we recall the results obtained about the application of the model checker SATMC to the analysis and verification of models carrying LTL properties.

In Section 3, we describe the investigation a novel technique for vulnerability-driven test case generation mainly dealing with attack traces prioritization (Section 3.1). In particular, we present a new approach for obtaining attack traces according to an ordering relation. Ordered attack traces can be exploited for prioritizing test cases. Hence, a suitable model mutation procedure is described in Section 3.2. Mutated models are subsequently processed through SATMC (cf. Section 3.3) in order to obtain corresponding abstract attack traces.

2 Property-driven test case generation

In this section we report the advanced techniques for property-driven test case generation. Most emphasis is about *linear temporal logic* which is used for the formalization of security goals.

2.1 LTL separation

Linear temporal logic [19, 20], abbreviated to LTL, is a common language for specifying, and reasoning about, the behavior of reactive systems.

We introduce the notion of *canonical separations* in order to strengthen Gabbay's LTL separation theorem [10]: Gabbay shows that any LTL formula is semantically equivalent to a Boolean combination of finitely many LTL formulae, each of which refers either strictly to the future, strictly to the present, or strictly to the past. We show that any formula in its separated form can be transformed into a syntactically uniform formula. That is, there is a canonical separation for any LTL formula. This allows us to constructively prove that any LTL property can be decomposed into a safety property and a liveness property, both of them in LTL, thus extending the result of Alpern and Schneider in [4] which shows that any LTL property can be decomposed into a safety property and a liveness property, both of them being ω-regular languages.

Related Work

Any execution of a reactive system can be seen as an infinite sequence of states, events, assertions, etc. A *property* is a set of executions. Two significant types of properties are safety and liveness properties, first introduced by Lamport [13], and later formalized by Alpern and Schneider [3]. Many system verification and validation methods are restricted to safety and liveness properties (safety properties can be tested, while liveness properties cannot), and safety and liveness require different proof techniques (see, e.g., [4, 14]).

Linear temporal logic, LTL [19, 20], is often used for reasoning about concurrent programs. Consequently, recognizing and characterizing safety and liveness in LTL, apart from its theoretical significance, has practical implications. Sistla [21, 22] characterized safety, stable, absolute liveness, fairness, and other properties in LTL without past temporal connectives, provided algorithms to recognize safety and liveness in LTL without past, and characterizes various properties in less expressive sub-logics of LTL. We characterize safety, stable, absolute liveness, fairness, and liveness, which Sistla left as an open problem, in LTL with past temporal operators, by reducing the problem of recognizing these properties to the satisfaction problem for LTL. We also adapt Sistla's formalization of stable

properties [22] to LTL with past temporal connectives. Lichtenstein et al. [15] characterize safety and liveness in LTL with past temporal operators, but they use a different notion of liveness than we do, which does not conform with the formalization by Alpern and Schneider [3].

A result of Alpern and Schneider [3] (Corollary 1.1) states that if a formalism $\mathcal{F}$ is closed under complement, intersection, and topological closure, then any property expressible in $\mathcal{F}$ is the intersection of an $\mathcal{F}$-expressible safety property and an $\mathcal{F}$-expressible liveness property. In [4], Alpern and Schneider show that ω-regular expressions (ie. Büchi automata [7]) satisfy these conditions and can therefore be decomposed as an intersection of a liveness ω-regular expression and a safety ω-regular expression. We constructively show that LTL is closed under topological closure. Since LTL is closed under intersection and complement (conjunction and negation, respectively), it then immediately follows that any property that is expressible as an LTL formula is the intersection of a safety and a liveness property, themselves expressible as LTL formulae. Lichtenstein et al. [15] use separation to rewrite LTL formulae in a safety-liveness normal form, but this does not decompose properties in the sense of Alpern and Schneider [3] since Lichtenstein et al. use a different formalization of liveness.

To prove these results, we use a proof technique based on Gabbay's separation theorem for LTL [10], which states that every LTL formula is semantically equivalent to a Boolean combination of finitely many LTL formulae, each of which refers either strictly to the future, strictly to the present, or strictly to the past. Moreover, Gabbay presents an algorithm for separation. We use this algorithm to construct, for any LTL formula φ, a formula we call the *canonical separation* of anchored φ, which we use to reduce the problem of recognizing safety, liveness, stable, absolute liveness, and fairness properties to the satisfaction problem for LTL. The canonical separation of anchored φ induces a right invariant equivalence relation $\approx_\varphi$ on the set of *traces*, i.e. finite executions. Finite traces t and u are equivalent, with respect to $\approx_\varphi$, if and only if for every infinite execution π either both $t\pi$ and $u\pi$ satisfy φ, or both falsify φ; cf. the Myhill-Nerode theorem for regular languages [17]. We use this equivalence relation to constructively prove that the topological closure of a property defined by an LTL formula is also definable by an LTL formula. This gives us an algorithm for decomposing LTL formulae into a conjunction of a safety LTL formula and a liveness LTL formula.

Road map

In Section 2.1.1, we introduce the syntax and semantics of LTL, and in Section 2.1.2 we recall Gabbay's separation theorem. In Section 2.1.3, we introduce safety and liveness, their topological characterizations, and the decomposition theorem for ω-regular properties. In Section 2.1.4 we introduce a right-invariant

equivalence relation on the set of all non-empty finite traces. In Section 2.1.5, we define the canonical separation and show its connection to the previously defined right invariant equivalence relation. In Section 2.1.6, we use this connection to prove that LTL is closed under topological closure and that it has a safety-liveness decomposition. In Section 2.1.7 we use canonical separation to characterize liveness and safety LTL properties, and in Section 2.1.8 we characterize stable, absolute liveness and fairness LTL properties.

2.1.1 Syntax and Semantics of LTL

We define the syntax and semantics of LTL with both past and future temporal connectives, initial and global equivalence of LTL formulae, the notions of past, present and future LTL formulae, and we recall Gabbay's separation theorem. Our definitions in this section are standard, e.g. see [10, 12, 16].

Definition 1 (LTL Syntax)**.** *Let AP be a finite set of atomic propositions. The syntax of LTL is given by the grammar*

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi \vee \varphi \mid \bullet\varphi \mid \bigcirc\varphi \mid \varphi \mathcal{S} \varphi \mid \varphi \mathcal{U} \varphi,$$

where $a \in AP$.

We use the usual syntactic sugars: $\bot$ stands for $\neg\top$, $\varphi \wedge \psi$ stands for $\neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi$ stands for $\neg\varphi \vee \psi$, $\Diamond\varphi$ stands for $\top \mathcal{U} \varphi$, $\Box\varphi$ stands for $\neg\Diamond\neg\varphi$, $\blacklozenge\varphi$ stands for $\top \mathcal{S} \varphi$, and $\blacksquare\varphi$ stands for $\neg\blacklozenge\neg\varphi$.

Let Σ^ω be the set of all countably infinite sequences over the alphabet $\Sigma = 2^{AP}$. Any element of Σ^ω is a **path**. We reserve the word **trace** for the elements of Σ^+, the set of finite non-empty words over Σ. For a path $\pi = p_0 p_1 p_2 \ldots$ we define its **prefix** π_i as the trace $p_0 p_1 \cdots p_i$, and we define its **suffix** π^i as the path $p_i p_{i+1} \ldots$. A **property** is a set of paths.

Definition 2 (LTL Semantics)**.** *For a path $\pi = p_0 p_1 p_2 \ldots$ and $i \in \mathbb{N}_0$, the* **satisfaction relation** *for LTL formulae is defined inductively over the formula structure:*

$$\begin{array}{lll}
\pi, i \models \top & & \\
\pi, i \models a & \textit{if} & a \in p_i \\
\pi, i \models \neg\varphi & \textit{if} & \pi, i \not\models \varphi \\
\pi, i \models \varphi \vee \psi & \textit{if} & \pi, i \models \varphi \textit{ or } \pi, i \models \psi \\
\pi, i \models \bullet\varphi & \textit{if} & i > 0 \textit{ and } \pi, i-1 \models \varphi \\
\pi, i \models \bigcirc\varphi & \textit{if} & \pi, i+1 \models \varphi \\
\pi, i \models \varphi \mathcal{S} \psi & \textit{if} & \textit{there is a } j \leq i \textit{ such that } \pi, j \models \psi \\
 & & \textit{and } \pi, k \models \varphi \textit{ for all } j < k \leq i \\
\pi, i \models \varphi \mathcal{U} \psi & \textit{if} & \textit{there is a } j \geq i \textit{ such that } \pi, j \models \psi \\
 & & \textit{and } \pi, k \models \varphi \textit{ for all } i \leq k < j
\end{array}$$

When $\pi, i \models \varphi$, we say π **satisfies** *φ at* **time** *i. Any LTL formula φ defines a property $L(\varphi) = \{\pi \in \Sigma^\omega \mid \pi, 0 \models \varphi\}$.* □

In Section 2.1.2 we need the notions of past and future formulae. Below, we define these notions syntactically. Then, we show that the syntactical definitions satisfy the semantic conditions: any future (respectively past) formula is independent of the past (respectively the future).

Definition 3. *We define present, past, future, and strict future formulae:*

- *The syntax of* **present** *LTL formulae is given by the grammar*

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi \vee \varphi \text{ , where } a \in AP.$$

- *The syntax of* **past** *LTL formulae is given by the grammar*

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi \vee \varphi \mid \bullet\varphi \mid \varphi \mathcal{S} \varphi \text{ , where } a \in AP.$$

- *The syntax of* **future** *LTL formulae is given by the grammar*

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi \text{ , where } a \in AP.$$

- *If $\psi = \bigcirc\varphi$, where φ is a future formula, or $\psi \in \{\top, \bot\}$, we say that ψ is a* **strict future** *formula.*

□

We refer to the set of future LTL formulae as **FLTL**. Lemma 4 below intuitively states that the satisfaction of past formulae is independent of changing the future, the satisfaction of future formulae is independent of changing the past, and the satisfaction of strict future formulae is independent of changing the past or present.

The proof of the lemma is straightforward by induction on the structure of past, future and strict future formulae.

Lemma 4. *Let F be a future formula, P be a past formula and S be a strict future formula. Then for every $i \in \mathbb{N}_0$ and every path $\pi \in \Sigma^\omega$ we have:*

1. *$\pi, i \models F$ iff $t\pi^i, |t| \models F$, for every trace $t \in \Sigma^+$ of length $|t|$.*
2. *$\pi, i \models P$ iff $\pi_i\sigma, i \models P$, for every path $\sigma \in \Sigma^\omega$.*
3. *$\pi, i \models S$ iff $tx\pi_{i+1}, |t| \models S$, for any trace $t \in \Sigma^+$ and $x \in \Sigma$.*

□

Lemma 4 allows us to use the following notational conventions in the remainder of the paper: for a strict future formula S, we write $x\pi, 0 \models S$ to emphasize that the initial $x \in \Sigma$ can be arbitrary. For a past formula P and a trace $t \in \Sigma^+$, we write $t \models P$ for $t\pi, |t| - 1 \models P$, where π is arbitrary.

Now we define two equivalence relations on LTL formulae.

Definition 5. *Let φ and ψ be LTL formulae.*

1. **Global** *(or,* **semantical***) equivalence:*

$$\varphi \equiv_g \psi \;\; \textit{if} \;\; \forall \pi \in \Sigma^\omega.\, \forall i \in \mathbb{N}_0.\, \Big(\pi, i \models \varphi \iff \pi, i \models \psi\Big)$$

2. **Initial** *equivalence:*

$$\varphi \equiv_i \psi \;\; \textit{if} \;\; \forall \pi \in \Sigma^\omega.\, \Big(\pi, 0 \models \varphi \iff \pi, 0 \models \psi\Big)$$

□

Global equivalence implies initial equivalence; the converse does not hold: formulae a and $\blacksquare a$ are initially equivalent, but they are not globally equivalent.

It is easy to show that the two equivalence relations coincide for FLTL.

For every LTL formula φ there exists an FLTL formula ψ that is initially equivalent to φ; see [10, 11]. That is, φ and ψ define the same property, although the FLTL formula ψ is independent of the past.

In other words, past temporal connectives do not increase the expressiveness of FLTL, in the sense that LTL and FLTL define the same set of properties.

There are however several advantages to adding past temporal connectives to FLTL: specifications using both past and future temporal connectives are often more natural (e.g. see [8, 15]), and LTL formulae can be exponentially more succinct than their equivalent ones in FLTL [16].

2.1.2 LTL Separation

The separation theorem by Gabbay [10] states that every LTL formula is globally equivalent, through a sequence of re-writes, to a Boolean combination of pure past, pure present, and pure future formulae. Our present (respectively strict future) formulae coincide with Gabbay's pure present (respectively pure future) formulae. Our past formulae correspond to Boolean combinations of Gabbay's pure past and pure present formulae. Applying the distributive laws for conjunction and disjunction, we rephrase Gabbay's theorem in terms of a conjunction of implications.

Theorem 6 (Gabbay's LTL Separation)**.** *Each formula φ of LTL is globally equivalent, through a sequence of re-writes, to a formula of the form*

$$(P_1 \rightarrow F_1) \wedge \cdots \wedge (P_n \rightarrow F_n),$$

where P_i are past formulae and F_i are strict future formulae. □

We refer to this formula as a separated form or a **separation** of φ. We use the notation $(P \rightarrow F)_n$ to denote a separation consisting of n conjuncts. Each conjunct is an implication of the form $P_i \rightarrow F_i$, and we will refer to the formulae P_i and F_i as the **past part** and the **future part** of the conjunct, respectively.

Example 7. *Consider the simple precedence property $\varphi = \Box(a \rightarrow \blacklozenge b)$. This property may for instance specify that a person is granted access (denoted by a) to a resource only if they have been previously authorized to do so (denoted by b). The formula φ can be separated as:*

$$\Box(a \rightarrow \blacklozenge b) \equiv_g \Big(\blacksquare\neg b \rightarrow \bigcirc\neg\Big(\neg b\, \mathcal{U}\, (a \wedge \neg b)\Big)\Big) \wedge \Big(a \wedge \blacksquare\neg b \rightarrow \bot\Big)$$

Intuitively, the first conjunct of the separated form states that if the person was not authorized yet, then in the (strict) future it must not happen that they are granted access before they are authorized. This indicates that the property is not falsified in the strict future. The second conjunct states that unauthorized access is not granted in the present. △

On the non-Uniqueness of Separation

For an LTL formula φ there is no syntactically unique separation. As a trivial example, we can always add a conjunct $\top \rightarrow \top$. However, since all separations of a formula are semantically equivalent, whenever some past parts of two separations $(P \rightarrow F)_n$ and $(P' \rightarrow F')_m$ of the same formula are mutually satisfiable, their corresponding future parts must be semantically equivalent. More precisely,

Lemma 8. *Let $u \in \Sigma^+$ be an arbitrary trace. If the sets of indices $I = \{i \mid u \models P_i\}$ and $J = \{j \mid u \models P'_j\}$ are not empty, then $\bigwedge_{i\in I} F_i \equiv_g \bigwedge_{j\in J} F'_j$.*

Proof. Assume the contrary. Then, without loss of generality, there exists a path $x\pi \in \Sigma^\omega$ such that $x\pi, 0 \models \bigwedge_{i\in I} F_i$ and $x\pi, 0 \not\models \bigwedge_{j\in J} F'_j$. Therefore, for all indices $i \in \{1, \cdots, n\}$ either $u\pi, |u-1| \not\models P_i$ or $u\pi, |u-1| \models P_i \wedge F_i$. Thus $u\pi, |u-1| \models (P \rightarrow F)_n$ and $u\pi, |u-1| \models \varphi$. Since $x\pi, 0 \not\models \bigwedge_{j\in J} F'_j$, there is an index $j \in J$ such that $x\pi, 0 \not\models F'_j$. Therefore, $u\pi, |u-1| \not\models P'_j \rightarrow F'_j$, and $u\pi, |u-1| \not\models (P' \rightarrow F')_m$, which is a contradiction since a formula is semantically equivalent to its separation. □

This lemma is our first motivation for the construction of the canonical separated form which we define later in the paper. We will refer to this lemma again later, to prove that the canonical separation has unique structure.

2.1.3 Safety and Liveness

We now recall the notions of safety and liveness which were first introduced by Lamport [13]. Safety and liveness properties are interesting both from a theoretical and a practical point of view, for example in testing. In this section, we prove necessary and sufficient conditions for a language to allow decomposition into safety an liveness. Later in the paper we prove, using LTL separation, that LTL satisfies these conditions and therefore allows decomposition into safety and liveness.

Lamport defines a safety property as one that states that something (usually bad) will **not** happen. For example, the elevator door must not open while the elevator is between floors. A liveness property is defined as one that states that something (usually good) **must** eventually happen. For example, every process in a multitasking system will eventually be granted CPU time. There are several formalizations of safety and liveness, such as [2], but Alpern and Schneider's formalization [3] has become the widely accepted standard. They use a topological characterization of safety and liveness, attributed to Plotkin, to show that every property is an intersection of a safety and a liveness property.

Definition 9. *Let Σ be an alphabet and Σ^ω the set of all paths over Σ. A property is any set of paths.*

- *A set S of paths $\pi \in \Sigma^\omega$ is a* **safety** *property iff for every $\pi \in \Sigma^\omega$*

$$\pi \notin S \implies \exists i \in \mathbb{N}_0.\ \forall \sigma \in \Sigma^\omega.\ \pi_i\sigma \notin S.$$

  Intuitively, this states that for every path which is not in S, there exists a point in time at which it becomes **irremediable**. *This is the "bad" thing from Lamport's formulation of safety.*

- *A set L of paths $\pi \in \Sigma^\omega$ is a* **liveness** *property iff*

$$\forall t \in \Sigma^*.\ \exists \pi \in \Sigma^\omega.\ t\pi \in L.$$

  Intuitively, no matter what has already happened, it can be remedied in the future. It is only important that the "good" thing occurs.

□

These definitions are naturally extended to LTL: for an LTL formula φ, if $L(\varphi)$ is a safety (respectively liveness) property, we say that φ is safety (respectively liveness).

The Topology of Properties

Definition 10. *We define the* **topology of properties** *on* Σ^ω *as follows:*

- *For every trace* $t \in \Sigma^*$*, the set* $\{t\pi \mid \pi \in \Sigma^\omega\}$ *is an open set.*
- *The union of open sets is an open set.*
- *The intersection of finitely many open sets is an open set.*

Closed *sets are complements of open sets. A set is* **dense** *iff its intersection with any open set is non-empty. For a set* A*, we define its* **topological closure** $cl(A)$ *as the smallest closed set that contains* A*, i.e. the intersection of all closed sets containing* A*.* □

The topology of properties is interesting because safety properties correspond to closed sets and liveness properties correspond to dense sets. Therefore, the topological closure of a property P is the smallest safety property S such that $P \subseteq S$.

Theorem 11 (Safety-Liveness Decomposition [3])**.** *Let* $P \subseteq \Sigma^\omega$ *be an arbitrary set of paths. There exists a safety property* S *and a liveness property* L *such that* $P = S \cap L$*.* □

The decomposition theorem is straightforward to prove using the topology of properties: for any property P we take its topological closure to be S and for L we take $(\Sigma^\omega \backslash S) \cup P$.

Alpern and Schneider have shown in [4] that the topological closure of any ω-regular property (i.e. ω-regular language) is an ω-regular property and that there is a corresponding ω-regular liveness property. We remark that the result of [4] does not imply that every LTL formula is equivalent to a conjunction of a safety LTL formula and a liveness LTL formula. This is because LTL is strictly less expressive than ω-regular properties[25].

2.1.4 A right invariance equivalence relation on Σ^ω

Before we define the canonical separation, which we use to characterize safety, liveness, and other properties in LTL and show that LTL is closed under topological closure, we will discuss safety and liveness with respect to arbitrary properties in Σ^ω, not necessarily limited to LTL. For an arbitrary property $\mathcal{L}$, we define an equivalence relation $\approx_{\mathcal{L}}$ on the set of all finite traces. We then show how this relation can be used to characterize the topological closure of $\mathcal{L}$ and to recognize if $\mathcal{L}$ is a liveness property. In Section 2.1.6, we prove that the canonical separation of anchored φ can be used to define the equivalence classes of $\approx_{L(\varphi)}$, which allows us to apply the following results to LTL.

Definition 12. *We say that an equivalence relation on a set of traces Σ^+ is* **right invariant** *(with respect to concatenation) [17] when for all traces $u, v, z \in \Sigma^+$, the equivalence of u and v implies equivalence of uz and vz.* □

Definition 13. *Let $\mathcal{L}$ be a property. We define the equivalence relation $\approx_{\mathcal{L}}$ by*

$$u \approx_{\mathcal{L}} v \text{ if and only if } \forall \pi \in \Sigma^{\omega}.\ u\pi \in \mathcal{L} \iff v\pi \in \mathcal{L}.$$

□

It is immediate that $\approx_{\mathcal{L}}$ is right invariant, for any $\mathcal{L}$. Now let $v \in \Sigma^+$ be an irremediable trace with respect to the property $\mathcal{L}$, i.e. for every $\pi \in \Sigma^{\omega}$ let $v\pi \notin \mathcal{L}$. Then the set of all such traces is the equivalence class $[v]_{\approx_{\mathcal{L}}}$. We use the notation $\mathcal{L}_{\perp}$ to denote this set (which might be empty). We show that the topological closure of $\mathcal{L}$ and the condition for $\mathcal{L}$ to be a liveness property can be characterized using $\mathcal{L}_{\perp}$.

Lemma 14. *Let $\mathcal{L}$ be a property. Then, $cl(\mathcal{L}) = \{\pi \in \Sigma^{\omega} \mid \forall i \in \mathbb{N}_0.\ \pi_i \notin \mathcal{L}_{\perp}\}$.*

Proof. We show that $S = \{\pi \in \Sigma^{\omega} \mid \forall i \in \mathbb{N}_0.\ \pi_i \notin \mathcal{L}_{\perp}\}$ contains $\mathcal{L}$, that it is a safety property, and that it is a subset of every safety property containing $\mathcal{L}$. For a path $\pi \notin S$ there exists an $i \in \mathbb{N}_0$ such that $\pi_i \in \mathcal{L}_{\perp}$. Then $\pi_i\sigma \notin S$, for all paths σ. This proves that S is a safety property. Since $\pi_i \in \mathcal{L}_{\perp}$, in particular $\pi \notin \mathcal{L}$. Therefore, $\mathcal{L} \subseteq S$. Finally, assume there is a safety property S' containing $\mathcal{L}$ such that there is a path $\pi \in S$ that is not in S'. Since S' is a safety property, there exists an $i \in \mathbb{N}_0$ such that $\pi_i\sigma \notin S'$, and therefore also $\pi_i\sigma \notin \mathcal{L}$, for all $\sigma \in \Sigma^{\omega}$. It follows that $\pi_i \in \mathcal{L}_{\perp}$, but from the definition of S, $\pi_i \notin \mathcal{L}_{\perp}$, which is a contradiction. □

Corollary 15. *Let $\mathcal{F}$ be a set of properties. $\mathcal{F}$ is topologically closed if and only if*

$$\forall \mathcal{L} \in \mathcal{F}.\ \left\{\pi \in \Sigma^{\omega} \mid \forall i \in \mathbb{N}_0.\ \pi_i \notin \mathcal{L}_{\perp}\right\} \in \mathcal{F}.$$

□

The following lemma that characterizes liveness properties follows immediately from the definition of liveness.

Lemma 16. *A property $\mathcal{L}$ is a liveness property if and only if $\mathcal{L}_{\perp} = \emptyset$.* □

2.1.5 Canonical Separation

In this section, we define the notion of a **canonical separation** and establish a connection with the equivalence classes of $\approx_{L(\varphi)}$. This will allow us to apply the results of Section 2.1.4 to characterize liveness and safety in LTL, prove that the

"safety part" of an LTL formula is an LTL formula, and constructively prove the decomposition theorem for LTL.

For a given LTL formula φ, we will construct a new formula $\psi = (P_1 \rightarrow F_1) \wedge \cdots \wedge (P_n \rightarrow F_n)$, in separated form, such that:

(i) For every trace $t \in \Sigma^+$ there is a unique index i such that $t \models P_i$.

(ii) For every path $\pi \in \Sigma^\omega$, and every $i \in \mathbb{N}_0$, if $\pi, i \models \psi$ then $\pi, 0 \models \varphi$.

(iii) For every path $\pi \in \Sigma^\omega$, if $\pi, 0 \models \varphi$ then $\pi, i \models \psi$, for all $i \in \mathbb{N}_0$.

(iv) For all indices $i \neq j$, $F_i \not\equiv_g F_j$.

The intuition behind this construction is as follows. The first condition guarantees that ψ defines a partition on the set of finite traces. From the second condition, it follows that whenever $t \models P_i$, if a path π satisfies the corresponding future part F_i, the path $t\pi$ satisfies φ initially. The third condition ensures that whenever $t \models P_i$, if π falsifies the corresponding future part F_i, the path $t\pi$ falsifies φ initially. From these three conditions it follows that every set in the partition induced by ψ is contained in one of the equivalence classes of the relation $\approx_{L(\varphi)}$. Finally, the last condition ensures that these sets coincide with the equivalence classes.

We first observe that, in general, a separation of φ satisfies none of these conditions. We prove that the formula $\blacksquare\blacklozenge\varphi$, which we refer to as **anchored** φ satisfies (ii) and (iii). Thus, every separation of anchored φ also satisfies (ii) and (iii). We introduce the notion of a **canonical separation**, which is a separation that can be constructed from any separation, and satisfies (i) and (iv). Hence, a canonical separation of anchored φ satisfies (i)-(iv) and therefore partitions the set of finite traces into equivalence classes of $\approx_{L(\varphi)}$.

Example 17. *In this example, we show that a separation of our precedence property $\varphi = \square(a \rightarrow \blacklozenge b)$ does not satisfy (i) and (ii). Its separation is given by*

$$(a \wedge \blacksquare\neg b \rightarrow \bot) \wedge (\blacksquare\neg b \rightarrow \bigcirc\neg(\neg b \,\mathcal{U}\, (a \wedge \neg b))).$$

It is immediate that this formula does not satisfy (i). Let t be a trace of length $|t|$. A path $t\pi$ satisfies the separation at time $|t| - 1$ iff $t \models P_i$ implies $x\pi, 0 \models F_i$, for $i = 1, 2$. For example, we can append any path π to the trace $t = bbb$ to satisfy the separation, and thus also φ, **at time 2**. *This, however, does not imply that $t\pi, 0 \models \varphi$. We can append any path π to the trace $t = abb$ to satisfy the separation at time 2, but φ is falsified at time 0, i.e. $abb\pi, 0 \not\models \varphi$. As all separations of a formula φ are globally equivalent, it follows that no separation of φ satisfies (ii).* $\triangle$

Note that the conditions (ii) and (iii) can be expressed as the equivalence $\pi, 0 \models \varphi$ iff $\exists i.\ \pi, i \models \psi$ iff $\forall i.\ \pi, i \models \psi$. Therefore, every separation of such a formula ψ will satisfy (ii) and (iii) as well. Intuitively, this would allow us to reason, at time $|t|$ (i.e. the present time), about how a trace t can be extended to either satisfy or falsify φ at time 0, i.e. initially. We will use a method similar to how Fisher [9] defines his separated normal form. The idea is to "anchor" φ at time 0. By pretending $\blacksquare\blacklozenge$ we obtain a formula that evaluates at any time i to what φ evaluates at time 0.

Theorem 18. *For every LTL formula φ and path $\pi \in \Sigma^\omega$, the following are equivalent:*

1. $\pi, 0 \models \varphi$
2. $\exists i \in \mathbb{N}_0.\ \pi, i \models \blacksquare\blacklozenge\varphi$
3. $\forall i \in \mathbb{N}_0.\ \pi, i \models \blacksquare\blacklozenge\varphi$

Proof. 2 follows trivially from 3. If 2 holds, then $\pi, 0 \models \blacklozenge\varphi$, and $\pi, 0 \models \varphi$. If 1 holds, then for every $k \in \mathbb{N}_0$, $\pi, k \models \blacklozenge\varphi$. Therefore, for every $i \in \mathbb{N}_0$ and every $j \leq i$ it follows that $\pi, j \models \blacklozenge\varphi$. Then, by the definition of $\blacksquare$, for every $i \in \mathbb{N}_0$ it follows that $\pi, i \models \blacksquare\blacklozenge\varphi$. □

The implication $2 \rightarrow 1$ proves that anchored φ satisfies the condition (ii) and the implication $1 \rightarrow 3$ proves that it satisfies the condition (iii). As discussed before, this proves that $\pi, 0 \models \varphi$ and $\pi, i \models \blacksquare\blacklozenge\varphi$ are equivalent, for every $i \in \mathbb{N}_0$ and all paths π.

Example 19. *Returning to the formula $\varphi = \square(a \rightarrow \blacklozenge b)$, we now separate $\blacksquare\blacklozenge\varphi$:*

$$\blacksquare\blacklozenge\varphi \equiv_g \Big(\blacklozenge\big(a \wedge \blacksquare\neg b\big) \rightarrow \bot\Big) \wedge \Big(\blacksquare\neg b \rightarrow \bigcirc\neg\big(\neg b\, \mathcal{U}\, (a \wedge \neg b)\big)\Big)$$

Recall Example 7. The second conjunct of both separations coincides, but the first one is different: intuitively, in the formula above, we also "look back" to check that the property was not falsified in the past, while in the former example we only checked if it was falsified in the present or the future. We can use this formula to reason about satisfying continuations of traces, and to define the relation $\approx_{L(\varphi)}$, by defining its equivalence classes. We can see that $\approx_{L(\varphi)}$ will have three equivalence classes:

1. *$C_1 = L(\varphi)_\perp = \{t \in \Sigma^+ \mid t \models \blacklozenge(a \wedge \blacksquare\neg b)\}$ is the set of all irremediable traces.*

2. $C_2 = \{t \in \Sigma^+ \mid t \models \blacksquare(\neg b \wedge \neg a)\}$ *is the set of traces that can be extended to satisfy* φ *by paths in which* a *does not occur for the first time before* b *does.*

3. $C_3 = \{t \in \Sigma^+ \mid t \models \blacklozenge(b \wedge \neg\bullet\blacklozenge a)\}$ *is the set of all traces such that every continuation results in a path satisfying* φ.

It may be surprising that the past formulae defining the three classes do not correspond to the past formulae in the separation. This is because the separation does not satisfy condition (i) and the two past formulae in the separation are mutually satisfiable. For example, the trace a *satisfies both of them, it can be extended in such a way that the future formula of the second conjunct is satisfied, but the resulting path will not satisfy* φ*, since we cannot satisfy* $\bot$ *in the first conjunct. This motivates the definition of a (structurally) canonical separation. Intuitively, a canonical separation is a separation that satisfies conditions (i) and (iv), and therefore induces an equivalence relation on the set of finite traces* Σ^+. △

To construct the canonical separation of a formula ψ, we take any separation of ψ and apply the procedure explained below. This results in a new formula, denoted $(P \xrightarrow{c} F)_n$, which we call a canonical separation of ψ. This formula is also separation of ψ, and in the case of $\psi = \blacksquare\blacklozenge\varphi$, it satisfies conditions (i) and (iv) for the formula φ.

Definition 20. *Let* $(P \to F)_n$ *be a separation of an LTL formula* ψ*. Consider the formula*

$$\bigwedge_{I\subseteq\{1,\ldots,n\}} \left(\left(\left(\bigwedge_{i\in I} P_i\right) \wedge \left(\bigwedge_{j\in\{1,\ldots,n\}\setminus I} \neg P_j\right)\right) \to \bigwedge_{i\in I} F_i\right).$$

From this formula, eliminate all conjuncts with past parts that are globally unsatisfiable, i.e. such that $\Diamond P \equiv_i \bot$*, and combine all conjuncts with semantically equivalent future parts. We call the resulting formula, denoted* $(P \xrightarrow{c} F)_n$ *a* **canonical separation** *of* ψ. □

It is immediate that a canonical separation of ψ is a separation of ψ, that every trace $t \in \Sigma^+$ satisfies the past part of exactly one of its conjuncts, and that no two of its conjuncts have semantically equivalent future parts. Therefore, it is a separation of ψ and satisfies conditions (i) and (iv). To simplify further discussion, if the canonical separation contains no conjunct with an unsatisfiable future part, we add the conjunct $\bot \to \bot$. Note that in the remainder of the paper, for a formula φ we wish to reason about, we are generally interested in a canonical separation of *anchored* φ.

Theorem 21. *Let let* $(P \xrightarrow{c} F)_n$*, and* $(P' \xrightarrow{c} F')_m$ *be two canonical separations of anchored formula* φ*. Then, they have the same structure, namely:*

- *$n = m$, i.e. they have the same number of conjuncts.*
- *For every index i there is an index j such that $P_i \equiv_g P'_j$ and $F_i \equiv_g F'_j$.*

Proof. Choose a conjunct $P \to F$ other than $\bot \to \bot$ from one of the canonical separations. Since P is satisfiable, there is a trace $u \in \Sigma^+$ such that $u \models P$. By the condition (i), u does not satisfy the past part of any other conjunct in the first canonical separation, and there is exactly one conjunct $P' \to F'$ in the second canonical separation such that $u \models P'$. Since both formulae are separations of the same initial formula, it follows from (8) that $F \equiv_g F'$.

Now assume one of the canonical separations does not have the conjunct $\bot \to \bot$. Then it must have a conjunct $P \to F$ such that P is satisfiable and $F \equiv_g \bot$. From the proof so far, it follows that the other canonical separation has such a conjunct. Therefore, by the construction of the canonical separation, none of the formulae have the conjunct $\bot \to \bot$. □

Example 22. *Let us return to our running example $\varphi = \Box(a \to \blacklozenge b)$. We take the separation from Example 19, and use it to construct a canonical separation of anchored φ:*

$$\Big(\blacklozenge\big(a \wedge \blacksquare\neg b\big) \to \bot\Big) \wedge$$
$$\Big(\blacksquare\big(\neg b \wedge (\neg a \vee \blacklozenge b)\big) \to \bigcirc\neg\Big(\neg b\,\mathcal{U}\,(a \wedge \neg b)\Big)\Big) \wedge$$
$$\Big(\big(\blacklozenge b \wedge \blacksquare(\neg a \vee \blacklozenge b)\big) \to \top\Big)$$

We can observe that the number of conjuncts corresponds to the number of equivalence classes of the relation $\approx_{L(\varphi)}$, where $L(\varphi)$ is the property specified by φ, i.e. the set of all paths satisfying φ at the initial time. Each class is defined as the set of all traces that satisfy the past formula of the same conjunct. Note that the past part of the second conjunct differs from the formula in the definition of class C_2 in Example 19. This is not a problem because the two formulae are semantically equivalent. Since $L(\varphi)_\bot \neq \emptyset$, we can conclude that the property specified by φ is not a liveness property. △

To simplify further discussion, for a canonical separation $(P \xrightarrow{c} F)_n$ we use a distinguished index $\bot$ to refer to the conjunct such that $F_\bot \equiv_g \bot$. For a formula φ, and any canonical separation of anchored φ, we define the sets

$$C_i = \big\{t \in \Sigma^+ \mid t \models P_i\big\}.$$

Theorem 21 ensures these sets are independent of the particular choice of canonical separation (allowing for permutation of indices). For an index i and a trace

$t \in C_i$, the path $t\pi$ satisfies formula φ initially iff $x\pi, 0 \models F_i$. Thus, if $t' \in C_i$, it follows that $t \approx_{L(\varphi)} t'$. Since in the construction of a canonical separation we do not allow two conjuncts with semantically equivalent future parts, we have the following theorem:

Theorem 23. *Let φ be an LTL formula and let $L(\varphi)$ be the property specified by φ. The equivalence classes of $\approx_{L(\varphi)}$ are given by the sets C_i. Furthermore, $L(\varphi)_\perp = C_\perp$.* □

Applications of the Canonical Separation

Theorem 23 allows us to apply the results from Section 2.1.4 to LTL and canonical separation. We prove that LTL is closed under topological closure and show how to construct the closure of a formula using canonical separation. This allows us to prove LTL formulae can be decomposed into safety and liveness in LTL. We show that deciding if an LTL formula φ is liveness is equivalent to the unsatisfiability of $\Diamond P_\perp$ and that deciding if a formula φ is safety is equivalent to the unsatisfiability of $\neg\varphi \wedge \Box\neg P_\perp$. Finally, we recall the notions of stable, absolute liveness, and fairness properties, and use canonical separation of anchored φ to characterize them.

2.1.6 Closure and Decomposition

Theorem 24. *The topological closure of a property $L(\varphi)$ specified by an LTL formula φ is the property $L(\Box\neg P_\perp)$ specified by the LTL formula $cl(\varphi) := \Box\neg P_\perp$.*

Proof. By Lemma 14, we must prove that $\{\pi \in \Sigma^\omega \mid \pi, 0 \models \Box\neg P_\perp\}$ is equal to $\{\pi \in \Sigma^\omega \mid \forall i \in \mathbb{N}_0.\ \pi_i \notin L(\varphi)_\perp\}$. This is straightforward: $\pi, 0 \models \Box\neg P_\perp$ iff $\forall i \in \mathbb{N}_0.\ \pi, i \models \neg P_\perp$ iff $\forall i \in \mathbb{N}_0.\ \pi, i \not\models P_\perp$ iff $\forall i \in \mathbb{N}_0.\ \pi_i \not\models P_\perp$ iff $\forall i \in \mathbb{N}_0.\ \pi_i \notin C_\perp$ iff $\forall i \in \mathbb{N}_0.\ \pi_i \notin L(\varphi)_\perp$. □

For the formula $cl(\varphi)$, we say it is the **safety part**, or closure, of formula φ. Since this construction can be done for every LTL formula, it follows that the set of LTL properties satisfies (15), which proves LTL is closed under topological closure.

Corollary 25. *LTL is closed under topological closure.* □

Now that we have shown that LTL is closed under topological closure, it is simple to show that LTL formulae can be decomposed. We construct the **liveness part** following Alpern and Schneider in [3].

Lemma 26. *Let φ be an LTL formula. Then the formula $live(\varphi) = \varphi \vee \neg cl(\varphi)$ is liveness.*

Proof. Assume the contrary. Then there is a trace t such that for every path π the formula $live(\varphi)$ is not satisfied by $t\pi$ at time 0. Therefore, for every path π also $t\pi, 0 \not\models \varphi$ and $t\pi, 0 \models cl(\varphi)$. The latter implies $t \notin C_\perp$. Hence, there is a path σ such that $t\sigma, 0 \models \varphi$, which is a contradiction. □

We can finally prove the decomposition theorem for LTL. This is now completely straightforward, since by applying distributive laws $cl(\varphi) \wedge live(\varphi) = cl(\varphi) \wedge (\varphi \vee \neg cl(\varphi)) \equiv_g \varphi$. This is an application of Corollary 1.1 from [3].

Corollary 27 (The decomposition theorem for LTL)**.** *Every formula φ of LTL is semantically equivalent to the conjunction of LTL formulae $cl(\varphi) \wedge live(\varphi)$, where $cl(\varphi)$ is a safety LTL formula and $live(\varphi)$ is a liveness LTL formula.* □

To complete the picture in a sense, we also mention that every LTL formula can be expressed as a conjunction of two liveness LTL formulae. More precisely,

Corollary 28. *Every formula φ of LTL is semantically equivalent to the conjunction of two liveness LTL formulae.*

Proof. As a result of Corollary 2.1. from [3], we only have to show that there exist two liveness LTL formulae φ and ψ such that $\varphi \wedge \psi \equiv_i \perp$. We can take $\varphi = \Box\Diamond a$ and $\psi = \neg\varphi \equiv_g \Diamond\Box\neg a$. It is easy to show both formulae are liveness, and their conjunction is not satisfiable. □

2.1.7 Characterizing Liveness and Safety

Let φ be an LTL formula and let $(P \xrightarrow{c} F)_n$ be a canonical separation of anchored φ. As before, let $P_\perp$ be the past part of the conjunct $P_\perp \rightarrow F_\perp$ such that $F_\perp \equiv_g \perp$. Note that, since F_i are future formulae, this is equivalent to $F_\perp \equiv_i \perp$, i.e. that $F_\perp$ is not satisfiable.

Theorem 29 (Characterization of Liveness)**.** *A formula φ is liveness iff the formula $\Diamond P_\perp$ is not satisfiable, i.e. $\Diamond P_\perp \equiv_i \perp$.*

Proof. From (16), and the connection between $C_\perp$ and the definition of $P_\perp$, it follows that φ is liveness iff every trace $t \in \Sigma^+$ falsifies $P_\perp$, i.e. $t \not\models P_\perp$. This is equivalent to the condition that, for every path $\pi \in \Sigma^\omega$ and every $i \in \mathbb{N}_0$ also $\pi, i \not\models P_\perp$, which is equivalent to $\pi, i \models \neg P_\perp$. Thus, φ is liveness iff for every path π, we have $\pi, 0 \models \Box\neg P_\perp$. This is equivalent to $\pi, 0 \models \neg\Diamond P_\perp$ and finally $\pi, 0 \not\models \Diamond P_\perp$, for all paths $\pi \in \Sigma^\omega$. This is equivalent to $\Diamond P_\perp \equiv_i \perp$. □

Note that all conjuncts with such a past part are actually removed during the construction of the canonical separation. It follows that $(P \xrightarrow{c} F)_n$ must have the

conjunct $\bot \rightarrow \bot$ and therefore $P_\bot = \bot$. This also establishes, for LTL, a well-known result from [3] that a property is liveness iff its topological closure is the set of all paths, since $cl(\varphi) = \Box\neg P_\bot = \Box\neg\bot \equiv_g \Box\top \equiv_i \top$.

Corollary 30. *A formula φ is liveness if and only if $cl(\varphi) \equiv_i \top$.* □

A formula φ is safety if and only if the property $L(\varphi)$ is equal to $L(\Box\neg P_\bot)$. In other words, φ is safety iff $\varphi \equiv_i cl(\varphi)$. From (24) it follows that $L(\varphi) \subseteq L(\Box\neg P_\bot)$, therefore, it is enough to show that there is no path $\pi \in \Sigma^\omega$ such that $\pi, 0 \models \Box\neg P_\bot$ and $\pi, 0 \not\models \varphi$. This gives us the following characterization of safety.

Theorem 31 (Characterization of Safety)**.** *A formula φ is safety iff the formula $\neg\varphi \wedge \Box\neg P_\bot$ is not satisfiable, i.e. $\neg\varphi \wedge \Box\neg P_\bot \equiv_i \bot$.* □

To illustrate these results, we will once again refer to our running example.

Example 32. *Let us take another look at our running example $\varphi = \Box(a \rightarrow \blacklozenge b)$. In (22), we gave a canonical separation of anchored φ. It is immediate that $P_\bot = \blacklozenge(a \wedge \blacksquare\neg b)$.*

Observe the formula $\Diamond\blacklozenge(a \wedge \blacksquare\neg b)$. This formula is satisfiable, for example by the path a^ω. Therefore, φ is not liveness.

We can explicitly construct the safety part of φ:

$$cl(\varphi) = \Box\neg\blacklozenge(a \wedge \blacksquare\neg b)$$

This formula is semantically equivalent to $\Box\blacksquare(a \rightarrow \blacklozenge b)$, and initially equivalent to $\Box(a \rightarrow \blacklozenge b)$. Therefore, the formula $\neg\varphi \wedge \neg cl(\varphi)$ is initially equivalent to $\neg\Box(a \rightarrow \blacklozenge b) \wedge \Box(a \rightarrow \blacklozenge b)$. This formula is not satisfiable, and therefore initially equivalent to $\bot$, which proves that φ is safety. △

2.1.8 Characterizing Stability, Absolute Liveness, and Fairness

The notion of fairness was formalized by Sistla [22]. He defines a fairness property as a property that is both stable and absolute liveness. Informally, a fairness property can neither be satisfied nor falsified in finite time.

Definition 33. *Let φ be an LTL formula.*

1. *φ is* **stable** *if for every path π, initial satisfaction $\pi, 0 \models \varphi$ implies $\pi^i, 0 \models \varphi$, for all $i \in \mathbb{N}_0$.*

2. *φ is* **absolute liveness** *if φ is satisfiable, and for all traces t and every path π such that $\pi, 0 \models \varphi$, also $t\pi, 0 \models \varphi$.*

3. *φ is* **fairness** *if it is both stable and absolute liveness.*

□

Note that we adapted Sistla's formalization of stable properties to LTL with past connectives. In [22], φ is stable if $\pi, 0 \models \varphi$ implies $\pi, i \models \varphi$, for every i. However, this definition assumes φ is a formula of FLTL. The definitions coincide when φ is a future formula, but otherwise they do not: consider the formula $\blacksquare\blacklozenge a$. This formula is not stable with our definition, but would be if we used Sistla's with our syntax. Our modification formalizes the notion of a stable property as a property containing all suffixes of its elements in LTL.

Let φ be an LTL formula and let $(P \xrightarrow{c} F)_n$ be a canonical separation of anchored φ. We first show necessary and sufficient conditions for φ to be stable and absolute liveness, respectively. We combine the two in a characterization of fairness and conclude the section with an example. In the following characterizations, we will use the fact that every LTL formula is initially equivalent to a formula of FLTL [10, 11]. For a formula φ, we will denote such an equivalent formula by φ^F.

Lemma 34. *A formula φ is stable iff the formula $\Psi = \bigvee_{i,j=1}^{n} \Diamond(P_i \wedge P_j^F \wedge F_i \wedge \neg F_j)$ is not satisfiable.*

Proof. Let φ be stable and assume the contrary, i.e. there is a path $\pi \in \Sigma^\omega$ such that $\pi, 0 \models \Psi$. Then there exists a $k \geq 0$ and indices i, j such that $\pi, k \models P_i$, $\pi, k \models P_j^F$, $\pi, k \models F_i$, and $\pi, k \models \neg F_j$. Since $\pi, k \models P_i$ and $\pi, k \models F_i$, it follows that $\pi, 0 \models \varphi$. From $\pi, k \models P_j^F$, it follows that $\pi^k, 0 \models P_j$. Since $\pi, k \models \neg F_j$, also $\pi^k, 0 \models \neg F_j$, and $\pi^k, 0 \not\models \varphi$. This is a contradiction with the assumption that φ is stable.

To prove the converse, let Ψ not be satisfiable, and assume φ is not stable. Then, there exists a path π such that $\pi, 0 \models \varphi$ and a $k > 0$ such that $\pi^k, 0 \not\models \varphi$. Let i and j be indices such that $\pi_k \models P_i$ and $p_k \models P_j$. It follows that $\pi^k, 0 \models F_i$ and $\pi^k, 0 \not\models F_j$. Therefore, $\pi, k \models P_i \wedge P_j^F \wedge F_i \wedge \neg F_j$, and $\pi, 0 \models \Psi$, which is a contradiction.

□

Example 35. *Let $\varphi = \Box a$. The formula $\blacksquare\blacklozenge\Box a$ is semantically equivalent to $\blacksquare\Box a$. From this, it is easy to see $(\blacklozenge\neg a \rightarrow \bot) \wedge (\top \rightarrow \bigcirc \Box a)$ is a separation of anchored φ. Therefore, a canonical separation of anchored φ is given by*

$$(\blacklozenge\neg a \rightarrow \bot) \wedge (\blacksquare a \rightarrow \bigcirc \Box a).$$

To prove φ is a stable formula, we need to show that the formulae $\Diamond(\blacklozenge\neg a \wedge a \wedge \bot \wedge \neg\bigcirc \Box a)$ and $\Diamond(\blacksquare a \wedge \neg a \wedge \bigcirc \Box a \wedge \top)$ are not satisfiable. This is immediate because of the $\bot$ in the first formula and $\blacksquare a \wedge \neg a$ in the second formula. Therefore $\Box a$ is a stable formula. △

Lemma 36. *Formula φ is absolute liveness iff φ is satisfiable and the formula $\Phi = \bigvee_{i,j=1}^{n} \Diamond(P_i \wedge P_j^F \wedge \neg F_i \wedge F_j)$ is not satisfiable.*

Proof. Let φ be absolute liveness. By the definition of absolute liveness, φ is satisfiable. Assume there is a path $\pi \in \Sigma^\omega$ such that $\pi, 0 \models \Phi$. Then there exists a $k \geq 0$ and indices i, j such that $\pi, k \models P_i$, $\pi, k \models P_j^F$, $\pi, k \models \neg F_i$, and $\pi, k \models F_j$. Analogously to the proof of the previous lemma, $\pi^k, 0 \models \varphi$ and $\pi, 0 \not\models \varphi$. This is a contradiction with the assumption that φ is absolute liveness.

To prove the converse, let Φ not be satisfiable, and assume φ is satisfiable but not absolute liveness. Then there exists a path π and a $k > 0$ such that $\pi^k, 0 \models \varphi$ and $\pi, 0 \not\models \varphi$. Let i and j be indices such that $\pi_k \models P_i$ and $p_k \models P_j$. It follows that $\pi^k, 0 \not\models F_i$ and $\pi^k, 0 \models F_j$. Therefore, $\pi, k \models P_i \wedge P_j^F \wedge \neg F_i \wedge F_j$, and $\pi, 0 \models \Phi$, which is a contradiction. □

Example 37. *Let $\varphi = \Diamond a$. The formula $\blacksquare\neg a \rightarrow \Diamond a$ is a separation of anchored φ. Therefore, a canonical separation of anchored φ is given by*

$$(\bot \rightarrow \bot) \wedge (\blacklozenge a \rightarrow \top) \wedge (\blacksquare\neg a \rightarrow \bigcirc \Diamond a).$$

It is immediate that φ is a liveness formula, and therefore satisfiable. To prove it is absolute liveness, we need to show that Φ is not satisfiable. Since $P_\bot = \bot$, we only need to consider indices $i, j \neq \bot$. The two formulae we still need to consider are $\Diamond(\blacklozenge a \wedge \neg a \wedge \bot \wedge \bigcirc \Diamond a)$ and $\Diamond(\blacksquare\neg a \wedge a \wedge \neg\Diamond a \wedge \top$. The first formula is not satisfiable because of the conjunct $\bot$, and the second formula is not satisfiable because of the conjunction $\blacksquare\neg a \wedge a$. Therefore, $\Diamond a$ is an absolute liveness formula. △

Theorem 38 (Characterization of fairness)**.** *A formula φ is fairness iff φ is satisfiable and there is an index i such that $\Diamond\neg P_i \equiv_i \bot$.*

Proof. Let φ be fairness. In particular, φ is liveness, satisfiable, and $\Diamond P_\bot \equiv_i \bot$. Assume the contrary, i.e that for all indices $i, j \neq \bot$ there are traces $u, v \in \Sigma^+$ such that $u \models P_i$ and $v \models P_j$. By the construction of canonical separation, F_i is satisfiable, and let $x\pi$ be a path such that $x\pi, 0 \models F_i$. Then $u\pi, 0 \models \varphi$. Since φ is stable, it follows that $\pi, 0 \models \varphi$. Since it is absolute liveness, $v\pi, 0 \models \varphi$. Therefore $x\pi, 0 \models F_j$. By repeating this proof with the indices exchanged, we get that F_i and F_j are semantically equivalent. This is a contradiction, since it is not possible by the construction of the canonical separation.

For the converse, the canonical separation satisfies the conditions of the previous lemmas and therefore φ is both stable and absolute liveness. □

Example 39. *Consider the property $\varphi = \Box\Diamond a$. The formula $\blacksquare\blacklozenge\Box\Diamond a$ is semantically equivalent to $\blacksquare\Box\Diamond a$, which is semantically equivalent to $\Box\Diamond a$. It is easy to*

Property	**Characterization (UNSAT)**
Safety	$\neg\varphi \wedge \Box\neg P_{\perp}$
Liveness	$\Diamond P_{\perp}$
• Absolute Liveness	$\bigvee_{i,j=1}^{n} \Diamond(P_i \wedge P_j^F \wedge \neg F_i \wedge F_j)$
Stable	$\bigvee_{i,j=1}^{n} \Diamond(P_i \wedge P_j^F \wedge F_i \wedge \neg F_j)$
Fairness	$\Diamond\neg P_i$, for some i

Figure 1: Reduction to UNSAT

see that $\top \rightarrow \bigcirc\Box\Diamond a$ *is one of its separations. Therefore, a canonical separation of anchored* φ *is given by*

$$(\perp \rightarrow \perp) \wedge (\top \rightarrow \bigcirc\Box\Diamond a).$$

This proves that φ *is a fairness property.* △

Theorem 40. *Recognizing safety, liveness, absolute liveness, safety, and fairness can be reduced to the non-satisfaction problem for LTL, summarized in Fig. (1).* □

A formula φ of LTL is safety, liveness, absolute liveness, safety, or fairness iff the respective formula in Fig. (1) is not satisfiable. Note that the characterization of absolute liveness assumes φ is liveness. This is justified since every absolute liveness formula is a liveness formula, and every liveness formula is satisfiable, and therefore satisfies the conditions of (36).

2.2 SATMC abstract attack trace generation

In this section we present the SATMC technique for model checking LTL properties. The complete description has been previously presented in [24]. Here we recall some excerpts of the technique, playing a key role for the generation of the attack trace in the test case generation.

2.2.1 SAT-reduction technique for First Order Linear Temporal Logic

Intuitively, the SATMC approach consists in reducing the problem of determining whether a security protocol violates a security property in $k > 0$ steps to the problem of checking the satisfiability of a propositional formula (the SAT problem). SAT encodings can be processed by means of efficient SAT solvers. By supporting the specification of security properties as (first-order) Linear Temporal Logic (LTL) formulae, SATMC is used for obtaining counterexample traces in the test case generation techniques described in Section 2.1.

At the core of our technique lies the construction of a data structure called *planning graph*. A planning graph provides a succinct representation of an over-approximation of the states that are reachable in k steps. In [5] the planning graph has been shown to generate concise encodings representing the set of execution paths. Such conciseness is crucial for the efficiency of our method. Below, we show how the planning graph is used for reducing any first-order LTL formula ϕ to an equivalent (in a sense that will be defined later) propositional formula ϕ_0 which is then reduced to SAT using available techniques (see, e.g., [6]).

The language of LTL we consider uses facts and equalities as atomic propositions, the usual propositional connectives ($\neg$, $\vee$, $\wedge$, $\Rightarrow$), the first-order quantifiers $\forall$ and $\exists$, and the temporal operators defined below. We denote with AP the set of atomic propositions. Let $\mathcal{V}$ be the set of variables used in the language and let $\mathcal{T}$ be the set of all ground terms that can be built by using the individual constants and the function symbols occurring in the specification of the system. An Herbrand interpretation is an interpretation $I_H = (D_H, g_h)$, where $D_H = \mathcal{T}$ and $g_H(f)(t_1, \ldots, t_n) = f(g_h(t_1), \ldots, g_h(t_n))$, i.e. $g_h(t) = t$ for all $t \in \mathcal{T}$. An Herbrand interpretation admits representation by means of the set of atomic formulae that are true in it. Thus, if p is an atomic formula we write $p \in I_H$ in place of $I_H \models p$. An *assignment over* $\mathcal{V}$ is a total function from $\mathcal{V}$ into $\mathcal{T}$, i.e. $\alpha : \mathcal{V} \to \mathcal{T}$. The extension of assignments to the set of facts is straightforward.

A *Kripke structure* is a tuple $M = (S, I, R, L)$, where S is the set of states, $I \subseteq S$ is the set of initial states, $R \subseteq (S \times S)$ is a total transition relation, i.e. for each $s \in S$ there exists $s' \in S$ such that $(s, s') \in R$, and $L : S \to 2^{AP}$ is a *labeling function*, i.e. a function that maps every state in an Herbrand interpretation. A *path* is an infinite sequence of states $\pi = s_0 s_1 s_2 \cdots$ such that $(s_i, s_{i+1}) \in R$ for all $i = 0, 1, \ldots$. If $\pi = s_0 s_1 s_2 \cdots$ is a path, with $\pi(i)$ we denote s_i and with π_i with denote the suffix $s_i, s_{i+1}, \ldots$ of π. We say that π is an *initialized path* iff $\pi_0 \in I$. Let π be an initialized path of M and α be an assignment over $\mathcal{V}$, an LTL formula ϕ *is satisfied by* α *in* π, written $\pi \models_\alpha \phi$, if and only if $\pi_0 \models_\alpha \phi$, where $\pi_i \models_\alpha \phi$, with $i \geq 0$, is inductively defined as follows:

$$
\begin{array}{ll}
\pi_i \models_\alpha f & \text{if } \alpha(f) \in L(\pi(i)) \ (f \text{ is a fact}) \\
\pi_i \models_\alpha (t_1 = t_2) & \text{if } \alpha(t_1) \text{ and } \alpha(t_2) \text{ are the same term} \\
\pi_i \models_\alpha \neg\phi & \text{if } \pi_i \not\models_\alpha \phi \\
\pi_i \models_\alpha (\phi_1 \vee \phi_2) & \text{if } \pi_i \models_\alpha \phi_1 \text{ or } \pi_i \models_\alpha \phi_2 \\
\pi_i \models_\alpha \mathbf{X}\,\phi & \text{if } \pi_{i+1} \models_\alpha \phi \\
\pi_i \models_\alpha \phi\,\mathbf{U}\,\psi & \text{if } \exists j \geq i.\pi_j \models_\alpha \psi \text{ and } \forall k \in [i,j).\pi_k \models_\alpha \phi \\
\pi_i \models_\alpha \phi\,\mathbf{R}\,\psi & \text{if } \forall j \geq i.\pi_j \models_\alpha \psi \text{ or } \exists k \in [i,j).\pi_k \models_\alpha \phi \\
\pi_i \models_\alpha \mathbf{Y}\,\phi & \text{if } i > 0 \text{ and } \pi_{i-1} \models_\alpha \phi \\
\pi_i \models_\alpha \phi\,\mathbf{S}\,\psi & \text{if } \exists j \in [0,i].\pi_i \models_\alpha \psi \text{ and } \forall k \in (j,i].\pi_k \models_\alpha \phi \\
\pi_i \models_\alpha \phi\,\mathbf{T}\,\psi & \text{if } \forall j \in [0,i].\pi_i \models_\alpha \psi \text{ or } \exists k \in (j,i].\pi_k \models_\alpha \phi \\
\pi_i \models_\alpha \exists x.\phi & \text{if } \pi_i \models_{\alpha[t/x]} \phi \text{ for some } t \in \mathcal{T}
\end{array}
$$

where $\alpha[t/x]$ is the assignment that associates x with t and all other variables y with $\alpha(y)$. Commonly LTL formulae exploit standards boolean shorthand $\top \equiv \phi \vee \neg\phi$, $\bot \equiv \neg\top$, $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, $\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$, $\forall x.\phi \equiv \neg\exists x.\neg\phi$ and the derived temporal operators $\mathbf{F}\,\phi \equiv \top\,\mathbf{U}\,\phi$ (finally), $\mathbf{G}\,\phi \equiv \neg\,\mathbf{F}\,\neg\phi$ (globally), $\mathbf{O}\,\phi \equiv \top\,\mathbf{S}\,\phi$ (once), $\mathbf{H} \equiv \bot\ \mathbf{T}\,\phi$ (historically). We use $\forall(\phi)$ and $\exists(\phi)$ as abbreviations of $\forall X_1 \ldots \forall X_n.\phi$ and $\exists X_1 \ldots \exists X_n.\phi$ respectively, where $X_1, \ldots, X_n$ are the free variables of the formula ϕ.

We say that ϕ *is valid in* M, in symbols $M \models \phi$, if and only if $\pi \models_\alpha \phi$ for all initialized paths π of M and all assignments α.

In rest of this section we show that planning graphs can be profitably used to turn any first-order LTL formula ϕ into a propositional LTL formula ϕ_0 such that if π is an execution path of M with k or less states that violates ϕ_0, then π violates also ϕ, and vice versa.

A planning graph is a sequence of layers Γ_i for $i = 0, \ldots, k$, where each layer Γ_i is a set of facts concisely representing a set of states $\|\Gamma_i\| = \{S : S \subseteq \Gamma_i\}$. The construction of a planning graph for a given model M goes beyond the scope of this section and the interested reader is referred to [5] for more details. For the purpose of this section it suffices to know that *(i)* Γ_0 is set to the initial state of M, *(ii)* if S is reachable from the initial state of M in i steps (i.e. i transitions in M), then $S \in \|\Gamma_i\|$ (or equivalently $S \subseteq \Gamma_i$) for $i = 0, \ldots, k$, and *(iii)* $\Gamma_i \subseteq \Gamma_{i+1}$ for $i = 0, \ldots, k-1$, i.e. the layers in the planning graph grow monotonically. From these key properties it is easy to see that if a fact does not occur in Γ_k, then it is false in all states reachable from the initial state in k steps and this readily leads to the following fact.

Fact 1. *Let ϕ be a propositional LTL formula, Γ_i for $i = 0, \ldots, k$ be a planning graph for M, and p a fact such that $p \notin \Gamma_k$. Then, $M \models_k \phi$ iff $M \models_k \phi[\bot/p]$, where $\phi[\bot/p]$ is the formula obtained from ϕ by replacing all occurrences of p with $\bot$.*

To illustrate let us consider the problem of generating a propositional version

of the following first-order formula:

$$\exists \mathtt{A}.\, \mathbf{F}(\neg\, \mathbf{O}\, s(\mathtt{A}, \mathtt{b}) \wedge r(\mathtt{b}, \mathtt{A})) \tag{1}$$

when $\Gamma_k = \{s(\mathtt{a1}, \mathtt{b}), r(\mathtt{b}, \mathtt{a1}), r(\mathtt{b}, \mathtt{a2})\}$.

If the variable $\mathtt{A}$ ranges over the (finite) set of constants $D_{\mathtt{A}} = \{\mathtt{a1}, \ldots, \mathtt{an}\}$, then we can replace the existential quantifier with a disjunction of instances of the formula in the scope of the quantifier, where each instance is obtained by replacing the quantified variable, namely $\mathtt{A}$, with the constants in $D_{\mathtt{A}}$:

$$\mathbf{F}(\neg\, \mathbf{O}\, s(\mathtt{a1}, \mathtt{b}) \wedge r(\mathtt{b}, \mathtt{a1})) \vee \ldots \vee \mathbf{F}(\neg\, \mathbf{O}\, s(\mathtt{an}, \mathtt{b}) \wedge r(\mathtt{b}, \mathtt{an})) \tag{2}$$

By repeatedly using Fact 1, formula (2) can be rewritten into:

$$\mathbf{F}(\neg\, \mathbf{O}\, s(\mathtt{a1}, \mathtt{b}) \wedge r(\mathtt{b}, \mathtt{a1})) \vee\ \mathbf{F}(\neg\, \mathbf{O}\, \bot \wedge r(\mathtt{b}, \mathtt{a2})) \vee\ \ldots \vee\ \mathbf{F}(\neg\, \mathbf{O}\, \bot \wedge \bot) \tag{3}$$

and finally be simplified to

$$\mathbf{F}(\neg\, \mathbf{O}\, s(\mathtt{a1}, \mathtt{b}) \wedge r(\mathtt{b}, \mathtt{a1})) \vee \mathbf{F}(r(\mathtt{b}, \mathtt{a2})) \tag{4}$$

Even if the resulting formula is compact thanks to the simplification induced by the planning graph, the instantiation step, namely the replacement of the existential quantifier with a disjunction of instances, can be a very expensive or even unfeasible (if the domain of the existentially quantified variable is not bounded).

A better approach is to let the instantiation activity be driven by the information available in the planning graph. This can be done by removing the existential quantification and generating instances by recursively traversing the remaining formula in a top down fashion. As soon as an atomic formula is met, the formula is matched against the facts in Γ_k and if a matching fact is found then the formula is replaced with the ground counterpart found in Γ_k and the corresponding matching substitution is carried over. The approach is iterated on backtracking so to generate all possible instances and when no (other) matching fact in Γ_k for the atomic formula at hand, then we replace it with $\bot$.

To illustrate, let us apply the approach to the formula (1). By removing the existential quantification we get the formula:

$$\mathbf{F}(\neg\, \mathbf{O}\, s(\mathtt{A}, \mathtt{b}) \wedge r(\mathtt{b}, \mathtt{A})) \tag{5}$$

By traversing (5) we find the atomic formula $s(\mathtt{A}, \mathtt{b})$ which matches with the fact $s(\mathtt{a1}, \mathtt{b})$ in Γ_k with matching substitution $\mathtt{A} = \mathtt{a1}$. The atomic formula $s(\mathtt{A}, \mathtt{b})$ is replaced with $s(\mathtt{a1}, \mathtt{b})$ in (5) and the constraint $\mathtt{A} = \mathtt{a1}$ is carried over. We are then left with the problem of finding a matching fact for the formula obtained

by applying the substitution to the right conjunct $r(\mathtt{b}, \mathtt{A})$, namely $r(\mathtt{b}, \mathtt{a1})$. The matching fact is easily found in Γ_k and we are therefore left with the formula

$$\mathbf{F}(\neg\, \mathbf{O}\, s(\mathtt{a1}, \mathtt{b}) \wedge r(\mathtt{b}, \mathtt{a1})) \tag{6}$$

as our first instance. By iterating the approach on backtracking we find that there is no other matching fact for $r(\mathtt{b}, \mathtt{a1})$ which is then replaced by $\bot$ thereby leading to the instance:

$$\mathbf{F}(\neg\, \mathbf{O}\, s(\mathtt{a1}, \mathtt{b}) \wedge \bot) \tag{7}$$

By further backtracking we find that there no other matching fact for $s(\mathtt{A}, \mathtt{b})$ in Γ_k and therefore we generate another instance by replacing $s(\mathtt{A}, \mathtt{b})$ with $\bot$ and carry over the constraint $\mathtt{A} \neq \mathtt{a1}$. The only fact in Γ_k matching $r(\mathtt{b}, \mathtt{A})$ while satisfying the constraint $\mathtt{A} \neq \mathtt{a1}$ is $r(\mathtt{b}, \mathtt{a2})$. We are then left with the formula

$$\mathbf{F}(\neg\, \mathbf{O}\, \bot \wedge r(\mathtt{b}, \mathtt{a2})) \tag{8}$$

as our third instance. No further matching fact for $r(\mathtt{b}, \mathtt{a2})$ exists and we thus obtain the formula

$$\mathbf{F}(\neg\, \mathbf{O}\, \bot \wedge \bot) \tag{9}$$

as our final instance. The procedure therefore turns the first-order LTL formula (1) into the disjunction of (6), (7), (8), and (9), which can in turn be simplified to (4).

3 Vulnerability-driven Test Case Generation

Below we present the advancements carried out in SPaCIoS for the vulnerability-driven test case generation. Mainly, we discuss the prioritization of abstract attack traces and how to obtain it through the extension of mutation operators and SATMC.

3.1 AAT prioritization

We implemented a set of low(high) level AVANTSSAR Specification Language (ASLan(++)) mutation operators to inject specific vulnerabilities into a secure specification (i.e., a specification for which no attack trace can be found by the model checkers). Each mutation operator is used to generate a set of modified specifications, called mutants, that, due to the injected vulnerability, are likely to lead to a violation of the defined security goals and therefore to an AAT. The set of attack traces generated this way is then used as a test suite for assessing the security of the target web application or protocol, i.e., the System Under Test (SUT).

More precisely, since every mutation operator injects a specific vulnerability, every time we succeed in reproducing one of the generated AATs on the implementation of the SUT, we know that the associated vulnerability is affecting the SUT's implementation. Given the different level of abstraction that characterizes an implementation and its specification in ASLan(++), in order to be able to execute an AAT on a real system we need to concretize it. This allows us to obtain the sequence of concrete inputs and expected outputs to use for probing the implementation of the SUT.

All the phases introduced so far have been already presented and discussed in details in previous deliverables ([23, 24]). In this document we focus exclusively on the latest developments based on combining model mutation, risk analysis, and model checking techniques to obtain a prioritized set of AATs. Having an indication about which of the test cases has the highest probability to expose a vulnerability in the SUT's implementation is important because the concretization and the execution of test cases is a resource consuming task.

3.2 Model mutation

The mutation strategy applied to obtain AAT's prioritization is similar to the one described in [18]. In that case, starting from the ASLan specification, we performed the following steps:

- we obtained the eXtensible Markup Language (XML) representation of the original ASLan model,
- we then applied a Extensible Stylesheet Language Transformation (XSLT) script implementing the mutation, and
- we finally got a set of ASLan mutated models by translating back the XML mutants into ASLan.

The difference from those passages and what we are presenting here consists in the fact that for prioritize the AATs (i) we have to introduce the weights coming from the risk analysis and (ii) we need to allow for the evaluation of all the weights coming from all possible mutations. To fulfill these objectives we implemented new mutating operators and modified those we have already implemented in order to deal with weights. We achieved (i) by assigning a weight to each mutation operator and to the mutated steps it introduces into the original specification by encoding the integer value in the name of the mutated step. This relation between operators and weights is given by the risk analysis which determines the likeliness of the presence of implementation errors, e.g., missing authorization checks, employment of a non-fresh value as nonce, which causes specific vulnerabilities

Listing 1: Excerpt of the NSPK_Lowe.aslan specification

```
step step_003_Initiator__line_16(E_S_IID, E_S_I_Actor, E_S_I_B, E_S_I_IID, Na,
    Na_1, Nb) :=
        child(E_S_IID, E_S_I_IID).
        not(dishonest(E_S_I_Actor)).
        state_Initiator(E_S_I_Actor, E_S_I_IID, 1, E_S_I_B, Na, Nb)
        =[exists Na_1]=>
        child(E_S_IID, E_S_I_IID).
        contains(E_S_I_Actor, secret_Na_set(E_S_IID)).
        contains(E_S_I_B, secret_Na_set(E_S_IID)).
        iknows(crypt(pk(E_S_I_B), pair(Na_1, E_S_I_Actor))).
        secret(Na_1, secret_Na, secret_Na_set(E_S_IID)).
        state_Initiator(E_S_I_Actor, E_S_I_IID, 3, E_S_I_B, Na_1, Nb)
```

in the SUT. In order to achieve (ii) we decided to exploit the model checking phase by providing SATMC with all the original and mutated step in a single mutant. This is the more significant difference, by an implementation point of view, from the way we used to mutate the specifications in [23, 24, 18]. It is also the reason why we had to create a new version of the already existing mutation operators as they have been implemented to generate a number of mutants equal to the number of possible way to inject one vulnerability in the given original specification. More precisely, they mutate the model by replacing one step with one of its possible mutated versions[1].

Indeed, the weight-based version of the operators creates one single mutants containing all the original steps plus a number of mutated steps equal to the number of mutants generated by the corresponding non weight-based version of the operator. This way, we provide SATMC with all possible additional mutated steps and, since all of them have a defined weight as they correspond to a vulnerability that has been injected, a way to order the AATs it is able to find. SATMC has been modified in order to report all possible AATs, and not only one of them as it usually did, and to manage the weights assigned to the mutated steps in order to prioritize the AATs (see Section 3.3 for more details).

To better show how the new weight-based mutation operators work, we give an example of mutation, highlighting the changes in the mutated steps, and all the details about the configuration file used in the process. In Listing 1 we reported one step belonging to the original ASLan specification of the NSPK protocol specification with the Lowe's fix (the entire model is available in Appendix B) on which we applied the "FreshnessFlaw" operator.

In the reported step agent `A`, which is the initiator of the protocol, sends to agent `B`, the responder, the message `{Na, A}_pk(B)`, where `Na` is a fresh value.

[1]Except for the Fact Assertion (FA) operator, and its semantic version, i.e., the Data Sanitization Flaw (DSFlaw) operator, described in [18]

This is the first message in every session of the protocol. By applying the FreshnessFlaw operator, we want to inject an error in the protocol, i.e., agent `A` does not send a fresh value, but the old value assigned to the respective variable `Na`. Therefore in the mutated step added by the mutation operator there is no `Na_1` on the RHS (i.e., no fresh value). In order apply this mutation, we have to set the configuration file in order to instruct the operator to remove the generation of fresh values for the variable `Na`. In Listing 2, we can see that the operator needs two inputs: the name of the variable we are going to target, and the weight assigned to the mutation.

Listing 2: Configuration file for the FreshnessFlaw operator

```
<FR>
  <variable>Na</variable>
  <weight>10</weight>
</FR>
```

While the first of the two is a regular expression (following the syntax rules in [1]) to identify the variables for which the generation of fresh values will be disabled, the second one is an integer representing the associated probability of the presence of the vulnerability injected by the operator in the SUT's implementation. With such a configuration, the FreshnessFlaw operator will look for all the steps having a string matching `Na` in the list of variable quantified with the **`exists`** keyword present between the LHS and the RHS, i.e., `=[exists Na_1]=>` in the step in Listing 1. For every such step, the operator will create a mutated version where `Na` is removed from the list of variables in the conditions (in the case it is the only one as in the example, the operator will remove entirely the existential quantification), and all occurrences of `Na_1` are replaced with `Na`. This mutated step is renamed with the concatenation of the original name with the string resulting from the concatenation of `_FR_`, the variable name in the removed condition, `_mw`, and the weight set in the configuration file. The result of the application of the FreshnessFlaw operator with the configuration file in Listing 2 is shown in Listing 3. Note that both versions of the step, the original and the mutated one, are present in the mutated specification; note also that the name of the original step is `step_003_Initiator__line_16` while the one of the mutated step is `step_003_Initiator__line_16_FR_Na_1_mw10`.

3.3 Prioritizing attack traces with SATMC

In this section we present how SATMC can deal with the prioritization of abstract attack traces. The approach documented below is still under investigation and further research and development are needed for implementing actual tools.

Listing 3: Excerpt of the mutated NSPK_Lowe.aslan specification

```
step step_003_Initiator__line_16(E_S_IID, E_S_I_Actor, E_S_I_B, E_S_I_IID, Na,
    Na_1, Nb) :=
        child(E_S_IID, E_S_I_IID).
        not(dishonest(E_S_I_Actor)).
        state_Initiator(E_S_I_Actor, E_S_I_IID, 1, E_S_I_B, Na, Nb)
        =[exists Na_1]=>
        child(E_S_IID, E_S_I_IID).
        contains(E_S_I_Actor, secret_Na_set(E_S_IID)).
        contains(E_S_I_B, secret_Na_set(E_S_IID)).
        iknows(crypt(pk(E_S_I_B), pair(Na_1, E_S_I_Actor))).
        secret(Na_1, secret_Na, secret_Na_set(E_S_IID)).
        state_Initiator(E_S_I_Actor, E_S_I_IID, 3, E_S_I_B, Na_1, Nb)

step step_003_Initiator__line_16_FR_Na_1_mw10(E_S_IID, E_S_I_Actor, E_S_I_B,
    E_S_I_IID, Na, Nb) :=
        child(E_S_IID, E_S_I_IID).
        not(dishonest(E_S_I_Actor)).
        state_Initiator(E_S_I_Actor, E_S_I_IID, 1, E_S_I_B, Na, Nb)
        =>
        child(E_S_IID, E_S_I_IID).
        contains(E_S_I_Actor, secret_Na_set(E_S_IID)).
        contains(E_S_I_B, secret_Na_set(E_S_IID)).
        iknows(crypt(pk(E_S_I_B), pair(Na, E_S_I_Actor))).
        secret(Na, secret_Na, secret_Na_set(E_S_IID)).
        state_Initiator(E_S_I_Actor, E_S_I_IID, 3, E_S_I_B, Na, Nb)
```

3.3.1 Enhancing the SATMC approach

In general, SATMC applies to the analysis of the protocol insecurity problem: i.e. running SATMC on a given protocol model, it determines whether there exists a trace violating the security requirements, i.e., an *attack trace*. In particular, since SATMC relies on a bounded search space representation, it looks for an attack trace of length less or equal to k, being k the search space limit. If such a trace exists, SATMC successfully reports it. Otherwise, SATMC ensures that no attack can be performed in less than k steps. We refer the reader to [5] for a detailed description.

Clearly, this approach does not allow analysts to obtain multiple attack traces from a single execution of SATMC. Moreover, if SATMC returns an attack trace, there is no guarantee that it is the only trace violating one of the goals, or, even more important, that it is the more relevant attack trace for that protocol Trace relevance and criticality are system specific. For instance, different attacks can affect and compromise different resources so impacting more or less disruptively on the system. In general, one could be interested in finding more than one attack per analysis and, possibly, order the attack traces according to a given priority scheme. Hence, SATMC needs to be extended with new features for (i) reporting a number of abstract attack traces of length less or equal to k and (ii) managing

the weights assigned by mutation operators (see Section 3.2) in order to achieve a concrete prioritization.

3.3.2 From SAT to Weighted Max-SAT

Given a positive integer k and a protocol specification, SATMC automatically generates a propositional formula by using sophisticated encoding techniques developed for planning; state-of-the-art SAT solvers taken off-the-shelf are then used to check the propositional formula for satisfiability

This approach makes it difficult to prioritize the found solution. Naively, we could consider the possibility of encoding information about trace priority directly in the SAT formula. Nevertheless, this solution would lead to extra, unmotivated computational overheads. Thus, we must reconsider the internal structure of SATMC.

We identified a possible solution in the application of *Weighted Max-SAT* solvers in place of standard SAT solvers. Briefly, the *maximum satisfiability problem* (Max-SAT) is the problem of finding an assignment of boolean variables which maximizes the number of clauses of a given formula. Among the variants to this problem, the *weighted Max-SAT* is one of the most studied in the literature.Intuitively, it consists in finding an assignment satisfying a boolean formula and maximizing the a set of weights. Weights are assigned to each clause of the boolean formula (in CNF format). Satisfied clauses contributes to the overall value of a solution by summing their weights.

SATMC uses a CNF encoding procedure for feeding the SAT-solver. Obtaining an encoding suitable for weighted Max-SAT consists in extending the current encoding step by adding weights annotations. Annotations appear in the encoding before each clause as in the following example.

```
p wcnf 3 4
10 1 -2 0
3 -1 2 -3 0
8 -3 2 0
5 1 3 0
```

Here, weights 10, 3, 8 and 5 are applied to the corresponding clauses.

Weights are directly extracted from the annotations injected through model mutation. Eventually, the output, i.e., a valid assignment, is mapped back to an abstract attack trace and its weight generates a corresponding trace priority.

3.3.3 Multiple attack traces generation

The approach described above allows for the generation of the abstract attack trace with the highest priority. Nevertheless, as mentioned in the previous sections, we

would like to generate a number of abstract attack traces ordered according to the priority. In order to obtain a new model, SATMC simply has to enrich the formula with a clause preventing the execution of (at least one of) the rules in the previous attack trace, and then it run again the solver. Alternatively, we can leverage the features of the state-of-the-art solvers. Indeed, given a formula, they natively support the generation of a number of models.

4 Conclusion

In this document, we presented the work carried out in the scope of the SPaCIoS project for the development of advanced test case generation techniques. Abstract test case generation is crucial for SPaCIoS since some crucial activities, e.g., test case execution, depend on it, e.g., for obtaining meaningful input. The investigation of new techniques improving and extending the current approaches is a main concern. Hence, in this document we focussed on presenting the research carried out for improving test case generation.

In this deliverable, we also identified new techniques and future directions. Among them, we focussed on LTL separation (Section 2) and attack trace prioritization (Section 3). The former provides a support for processing and decomposing LTL specifications. The latter is a useful mechanism for helping security analysts to find attack traces applying emphasis on certain, system specific issues. Since some of these aspects are still under investigation, we could not report practical results. Nevertheless, we identified technologies and tools providing the functionalities needed for their implementation.

A ASLan mutation operator disabling fresh nonces, `FreshnessFlaw_weights.xsl`

```
<?xml version="1.0" encoding="UTF-8" ?>

<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:xdt="http://www.w3.org/2005/xpath-datatypes"
  xmlns:err="http://www.w3.org/2005/xqt-errors"
  xmlns:f="http://www.example.com"
  exclude-result-prefixes="xs␣xdt␣err␣f␣fn">

  <xsl:output method="xml" indent="yes"/>

  <xsl:param name="config-file" as="xs:string" /> <!-- Location of the
      configuration file -->
  <xsl:param name="input-file" as="xs:string" /> <!-- Location of the input file
      -->
  <xsl:param name="output-dir" as="xs:string" /> <!-- Location of the input file
      -->

  <!-- Variable storing the original tree -->
  <xsl:variable name="main-doc" as="document-node()" select="/"/>

  <!-- Function returning a modified step -->
  <!-- it uses template in mode "mutate" which generates steps where the fresh --
      >
  <!-- generation part of the step is removed -->
  <xsl:function name="f:mutated_step" as="node()*">
    <xsl:param name="step-name" as="xs:string"/>
    <xsl:param name="nonce-name" as="xs:string"/>

    <xsl:variable name="out">
      <xsl:call-template name="remove_fresh">
        <xsl:with-param name="step" as="xs:string" select="$step-name" />
        <xsl:with-param name="nonceFresh" as="xs:string" select="$nonce-name" />
      </xsl:call-template>
    </xsl:variable>
    <xsl:copy-of select="$out/step"/>
  </xsl:function>

  <!-- Variable storing the mutated steps that will be added to the original XML
      tree -->
  <xsl:variable name="mutatedSteps" as="node()*">
    <!-- <xsl:message terminate="no">
      test== <xsl:value-of select="$main-doc/aslan/rules/step/exists/variable[
          matches(@name,document($config-file)/configuration/FR/variable)]"/>
    </xsl:message> -->
    <xsl:for-each select="$main-doc/aslan/rules/step/exists/variable[matches(
        @name,concat(document($config-file)/configuration/FR/variable,'_\d'))]">
      <xsl:sequence select="f:mutated_step(current()/parent::*/parent::*/@name,␣
          current()/@name)" />
      <!-- <xsl:message terminate="no">
        arguments to mutated_step = <xsl:value-of select="current()/parent::*/
            parent::*/@name" />
      </xsl:message> -->
    </xsl:for-each>
```

```
</xsl:variable>

<!-- main template which invokes the sub-templates defined below -->
<xsl:template match="/">
  <!-- <xsl:message terminate="no">
    mutatedSteps = <xsl:copy-of select="$mutatedSteps[1]" />
  </xsl:message> -->
  <xsl:variable name="uri" select="concat($output-dir,␣'/',␣substring-before($
      input-file,␣'.xml'),␣'_FR_weights',␣␣'.xml')"/>

  <xsl:result-document href="{$uri}">
    <xsl:apply-templates select="$main-doc" mode="once" />
  </xsl:result-document>
</xsl:template>

<!-- the identity/copy template -->
<xsl:template match="node(␣)␣|␣@*" mode="once">
  <xsl:copy>
    <xsl:apply-templates select="@*␣|␣node(␣)" mode="once"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="aslan/rules/step" mode="once">
  <xsl:copy-of select="."/>
  <xsl:variable name="stepNumber" select="substring(current()/@name,1,8)"/>
  <xsl:for-each select="$mutatedSteps">
    <xsl:choose>
      <!-- if step_XYZ original matches with the mutated one -->
      <!-- then we insert the mutated one -->
      <xsl:when test="substring(current()/@name,1,8)␣=␣$stepNumber">
        <xsl:copy-of select="."/>
      </xsl:when>
      <xsl:otherwise/>
    </xsl:choose>
  </xsl:for-each>
</xsl:template>

<xsl:template name="remove_fresh" as="node()">
  <xsl:param name="step" as="xs:string" />
  <xsl:param name="nonceFresh" as="xs:string" />

  <xsl:for-each select="$main-doc/aslan/rules/step[@name=$step]">
  <!-- First of all we copy the <step> tag and we nodify his attribute "name"
      -->
    <xsl:copy>
      <xsl:attribute name="name">
        <xsl:value-of select="concat(current()/@name,␣'_FR_'␣,␣$nonceFresh␣,␣'
            _mw',␣document($config-file)/configuration/FR/weight␣)"/>
      </xsl:attribute>
      <!-- then we process its children, i.e., comments, lhs, conditions,
          exists, rhs -->
      <!-- in order to mutate it by removing the creation of the fresh value --
          >
      <xsl:for-each select="child::node()">
      <xsl:choose>
        <!-- copying the parameters except the nonce and adjisting the position
            attribute -->
        <xsl:when test="name(.)='parameters'">
          <xsl:element name="parameters">
            <xsl:variable name="params" as="node()+">
              <xsl:for-each select="child::node()">
                <xsl:if test="not(current()/@name␣=␣$nonceFresh)">
```

```
                <xsl:sequence select="current()"/>
              </xsl:if>
            </xsl:for-each>
          </xsl:variable>

          <xsl:for-each select="$params">
            <xsl:copy>
              <xsl:attribute name="name">
                <xsl:value-of select="current()/@name"/>
              </xsl:attribute>
              <xsl:attribute name="position">
                <xsl:value-of select="floor(position() div 2)"/>
              </xsl:attribute>
            </xsl:copy>
          </xsl:for-each>
        </xsl:element>
      </xsl:when>

      <!-- removing nonce from exists -->
      <xsl:when test="name(.)='exists'">
        <xsl:choose>
          <xsl:when test="count(child::*/node()) = 1"/>
          <xsl:otherwise>
            <xsl:element name="exists">
              <xsl:for-each select="child::node()">
                <xsl:if test="not(current()/@name = $nonceFresh)">
                  <xsl:copy-of select="."/>
                </xsl:if>
              </xsl:for-each>
            </xsl:element>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:when>

      <!-- changing the rhs side -->
      <xsl:when test="name(.)='rhs'">
        <!-- <xsl:element name="rhs">
          <xsl:for-each select="descendant::node()"> -->
            <xsl:apply-templates select="." mode="remove">
              <xsl:with-param name="var" select="$nonceFresh" />
            </xsl:apply-templates>
        <!--  </xsl:for-each>
        </xsl:element>-->
      </xsl:when>

      <!-- copying what we do not want to change, e.g. lhs side, conditions
          -->
      <xsl:otherwise>
        <xsl:copy-of select="."/>
      </xsl:otherwise>
    </xsl:choose>
    </xsl:for-each>
  </xsl:copy>
</xsl:for-each>
</xsl:template>

<!--
  <xsl:template name="replace_fresh_var">
    <xsl:param name="var" />

    <xsl:choose>
      <xsl:when test="not(name(.)='variable') or not(current()/@name=$var)">
```

```
        <xsl:copy>
          <xsl:call-template name="replace_fresh_var">
            <xsl:with-param name="var" select="$var"/>
          </xsl:call-template>
        </xsl:copy>
      </xsl:when>
      <xsl:otherwise>
        <xsl:copy>
          <xsl:attribute name="name">
            <xsl:value-of select="substring(@name,1,string-length(@name)-2)"/>
          </xsl:attribute>
        </xsl:copy>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template> -->

  <!-- the identity/copy template modified to remove the last two characters from
       the fresh variable name -->
  <xsl:template match="node(␣)␣|␣@*" mode="remove">
    <xsl:param name="var" />
    <xsl:copy>
      <xsl:apply-templates select="@*␣|␣node(␣)" mode="remove">
        <xsl:with-param name="var" select="$var" />
      </xsl:apply-templates>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="@name" mode="remove">
    <xsl:param name="var"/>

    <xsl:attribute name="name">
      <xsl:choose>
        <xsl:when test=".␣=␣$var">
          <xsl:value-of select="substring(.,1,string-length(.)-2)"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="." />
        </xsl:otherwise>
      </xsl:choose>
    </xsl:attribute>
  </xsl:template>

</xsl:stylesheet>
```

B ASLan specification of the NSPK protocol with Lowe's fix

```
% @specification(NSPK_Lowe_Safe)
% @channel_model(CCM)
% @connector_name(ASLan++ Connector)
% @connector_version(1.4.7)
% @connector_options(-opt LUMP -hc ALL -gas)
% @satmc(--mutex=1)

section signature:

  text > slabel
  ak : agent -> public_key
  child : nat * nat -> fact
  ck : agent -> public_key
  defaultPseudonym : agent * nat -> public_key
  dishonest : agent -> fact
  hash : message -> message
  pk : agent -> public_key
  secret_Na_set : nat -> set(agent)
  secret_Nb_set : nat -> set(agent)
  state_Environment : agent * nat * nat -> fact
  state_Initiator : agent * nat * nat * agent * text * text -> fact
  state_Responder : agent * nat * nat * agent * text * text -> fact
  state_Session : agent * nat * nat * agent * agent -> fact
  succ : nat -> nat

section types:

  A : agent
  AM : message
  AR : agent
  AW : agent
  Actor : agent
  Ak_arg_1 : agent
  B : agent
  C : agent
  Ck_arg_1 : agent
  D : agent
  Dummy : agent
  % @original_name(name=Actor)
  E_S_Actor : agent
  % @original_name(name=IID)
  E_S_IID : nat
  % @original_name(name=Actor)
  E_S_I_Actor : agent
  % @original_name(name=B)
  E_S_I_B : agent
  % @original_name(name=IID)
  E_S_I_IID : nat
  % @original_name(name=SL)
  E_S_I_SL : nat
  % @original_name(name=A)
  E_S_R_A : agent
  % @original_name(name=A; match=true)
  E_S_R_A_1 : agent
  % @original_name(name=Actor)
  E_S_R_Actor : agent
  % @original_name(name=IID)
```

```
E_S_R_IID : nat
% @original_name(name=Na)
E_S_R_Na : text
% @original_name(name=Na; match=true)
E_S_R_Na_1 : text
% @original_name(name=Nb)
E_S_R_Nb : text
% @original_name(name=Nb; fresh=true)
E_S_R_Nb_1 : text
% @original_name(name=SL)
E_S_R_SL : nat
% @original_name(name=SL)
E_S_SL : nat
% @original_name(name=IID)
E_aIaR_IID : nat
% @original_name(name=AM)
E_aRaI_AM : message
% @original_name(name=AR)
E_aRaI_AR : agent
% @original_name(name=AW)
E_aRaI_AW : agent
% @original_name(name=IID)
E_aRaI_IID : nat
% @original_name(name=FM)
E_fRaI_FM : message
% @original_name(name=FR)
E_fRaI_FR : agent
% @original_name(name=FW)
E_fRaI_FW : agent
% @original_name(name=IID1)
E_fRaI_IID1 : nat
% @original_name(name=IID2)
E_fRaI_IID2 : nat
% @original_name(name=Knowers)
E_sN_Knowers : set(agent)
% @original_name(name=Msg)
E_sN_Msg : message
% @original_name(name=A)
E_siS_A : agent
% @original_name(name=B)
E_siS_B : agent
FM : message
FR : agent
FW : agent
Hash_arg_1 : message
IID : nat
IID1 : nat
IID2 : nat
% @original_name(name=IID; fresh=true)
IID_1 : nat
% @original_name(name=IID; fresh=true)
IID_2 : nat
% @original_name(name=IID; fresh=true)
IID_3 : nat
% @original_name(name=IID; fresh=true)
IID_4 : nat
Knowers : set(agent)
Msg : message
Na : text
% @original_name(name=Na; fresh=true)
Na_1 : text
Nb : text
```

```
  % @original_name(name=Nb; match=true)
  Nb_1 : text
  Pk_arg_1 : agent
  SL : nat
  Succ_arg_1 : nat
  atag : slabel
  auth_Initiator_authenticates_Responder : protocol_id
  auth_Responder_authenticates_Initiator : protocol_id
  ctag : slabel
  dummy_agent_1 : agent
  dummy_agent_2 : agent
  dummy_nat : nat
  dummy_text : text
  false : fact
  fresh_Initiator_authenticates_Responder : protocol_id
  fresh_Responder_authenticates_Initiator : protocol_id
  root : agent
  secret_Na : protocol_id
  secret_Nb : protocol_id
  stag : slabel
  % @original_name(name=C)
  symbolic_C : agent
  % @original_name(name=D)
  symbolic_D : agent
  % @original_name(name=A)
  symbolic_E_siS_A : agent
  % @original_name(name=B)
  symbolic_E_siS_B : agent
  true : fact

section inits:

% @new_instance(new_entity=Environment; Actor=root; IID=0; SL=1)
initial_state init :=
  child(dummy_nat, 0).
  dishonest(i).
  iknows(0).
  iknows(atag).
  iknows(ctag).
  iknows(i).
  iknows(inv(ak(i))).
  iknows(inv(ck(i))).
  iknows(inv(pk(i))).
  iknows(root).
  iknows(stag).
  iknows(symbolic_C).
  iknows(symbolic_D).
  iknows(symbolic_E_siS_A).
  iknows(symbolic_E_siS_B).
  state_Environment(root, 0, 1).
  true

section hornClauses:

hc public_ck(Ck_arg_1) :=
  iknows(ck(Ck_arg_1)) :-
    iknows(Ck_arg_1)

hc public_ak(Ak_arg_1) :=
  iknows(ak(Ak_arg_1)) :-
    iknows(Ak_arg_1)
```

```
hc public_pk(Pk_arg_1) :=
  iknows(pk(Pk_arg_1)) :-
    iknows(Pk_arg_1)

hc public_hash(Hash_arg_1) :=
  iknows(hash(Hash_arg_1)) :-
    iknows(Hash_arg_1)

hc public_succ(Succ_arg_1) :=
  iknows(succ(Succ_arg_1)) :-
    iknows(Succ_arg_1)

hc inv_succ_1(Succ_arg_1) :=
  iknows(Succ_arg_1) :-
    iknows(succ(Succ_arg_1))

section rules:

% @guard(entity=Environment; iid=IID; line=53; test=not(equal(E_siS_A, E_siS_B)))
% @new_instance(entity=Environment; iid=IID; line=53; new_entity=Session; Actor=
    dummy_agent_1; IID=IID_1; SL=1; A=E_siS_A; B=E_siS_B)
% @guard(entity=Environment; iid=IID; line=54; test=not(equal(C, D)))
% @new_instance(entity=Environment; iid=IID; line=54; new_entity=Session; Actor=
    dummy_agent_2; IID=IID_2; SL=1; A=C; B=D)
% @step_label(entity=Environment; iid=IID; line=54; variable=SL; term=3)
step step_001_Environment__line_53(Actor, C, D, E_siS_A, E_siS_B, IID, IID_1,
    IID_2) :=
  iknows(C).
  iknows(D).
  iknows(E_siS_A).
  iknows(E_siS_B).
  state_Environment(Actor, IID, 1) &
  not(equal(C, D)) &
  not(equal(E_siS_A, E_siS_B))
  =[exists IID_1, IID_2]=>
  child(IID, IID_1).
  child(IID, IID_2).
  iknows(C).
  iknows(D).
  iknows(E_siS_A).
  iknows(E_siS_B).
  state_Environment(Actor, IID, 3).
  state_Session(dummy_agent_1, IID_1, 1, E_siS_A, E_siS_B).
  state_Session(dummy_agent_2, IID_2, 1, C, D)

% @new_instance(entity=Session; iid=E_S_IID; line=40; new_entity=Initiator; Actor
    =A; IID=IID_3; SL=1; B=B; Na=dummy_text; Nb=dummy_text)
% @new_instance(entity=Session; iid=E_S_IID; line=41; new_entity=Responder; Actor
    =B; IID=IID_4; SL=1; A=A; Na=dummy_text; Nb=dummy_text)
% @step_label(entity=Session; iid=E_S_IID; line=41; variable=SL; term=3)
step step_002_Session__line_40(A, B, E_S_Actor, E_S_IID, IID_3, IID_4) :=
  not(dishonest(E_S_Actor)).
  state_Session(E_S_Actor, E_S_IID, 1, A, B)
  =[exists IID_3, IID_4]=>
  child(E_S_IID, IID_3).
  child(E_S_IID, IID_4).
  state_Initiator(A, IID_3, 1, B, dummy_text, dummy_text).
  state_Responder(B, IID_4, 1, A, dummy_text, dummy_text).
  state_Session(E_S_Actor, E_S_IID, 3, A, B)

% @fresh(entity=Initiator; iid=E_S_I_IID; line=16; variable=Na; term=Na_1)
% @introduce(entity=Initiator; iid=E_S_I_IID; line=16; fact=secret(Na_1,
```

```
    secret_Na, secret_Na_set(E_S_IID)))
% @introduce(entity=Initiator; iid=E_S_I_IID; line=16; fact=contains(E_S_I_Actor,
     secret_Na_set(E_S_IID)))
% @introduce(entity=Initiator; iid=E_S_I_IID; line=16; fact=contains(E_S_I_B,
    secret_Na_set(E_S_IID)))
% @communication(entity=Initiator; iid=E_S_I_IID; line=17; sender=E_S_I_Actor;
    receiver=E_S_I_B; payload=crypt(pk(E_S_I_B), pair(Na_1, E_S_I_Actor));
    channel=regularCh; fact=iknows(crypt(pk(E_S_I_B), pair(Na_1, E_S_I_Actor)));
    direction=send)
% @step_label(entity=Initiator; iid=E_S_I_IID; line=17; variable=SL; term=3)
step step_003_Initiator__line_16(E_S_IID, E_S_I_Actor, E_S_I_B, E_S_I_IID, Na,
    Na_1, Nb) :=
  child(E_S_IID, E_S_I_IID).
  not(dishonest(E_S_I_Actor)).
  state_Initiator(E_S_I_Actor, E_S_I_IID, 1, E_S_I_B, Na, Nb)
  =[exists Na_1]=>
  child(E_S_IID, E_S_I_IID).
  contains(E_S_I_Actor, secret_Na_set(E_S_IID)).
  contains(E_S_I_B, secret_Na_set(E_S_IID)).
  iknows(crypt(pk(E_S_I_B), pair(Na_1, E_S_I_Actor))).
  secret(Na_1, secret_Na, secret_Na_set(E_S_IID)).
  state_Initiator(E_S_I_Actor, E_S_I_IID, 3, E_S_I_B, Na_1, Nb)

% @communication(entity=Initiator; iid=E_S_I_IID; line=18; sender=E_S_I_B;
    receiver=E_S_I_Actor; payload=crypt(pk(E_S_I_Actor), pair(Na, pair(Nb_1,
    E_S_I_B))); channel=regularCh; fact=iknows(crypt(pk(E_S_I_Actor), pair(Na,
    pair(Nb_1, E_S_I_B)))); direction=receive)
% @match(entity=Initiator; iid=E_S_I_IID; line=18; variable=Nb; term=Nb_1)
% @introduce(entity=Initiator; iid=E_S_I_IID; line=18; fact=request(E_S_I_Actor,
    E_S_I_B, auth_Initiator_authenticates_Responder, Na, E_S_I_IID))
% @introduce(entity=Initiator; iid=E_S_I_IID; line=18; fact=request(E_S_I_Actor,
    E_S_I_B, fresh_Initiator_authenticates_Responder, Na, E_S_I_IID))
% @introduce(entity=Initiator; iid=E_S_I_IID; line=18; fact=secret(Nb_1,
    secret_Nb, secret_Nb_set(E_S_IID)))
% @introduce(entity=Initiator; iid=E_S_I_IID; line=18; fact=contains(E_S_I_Actor,
     secret_Nb_set(E_S_IID)))
% @introduce(entity=Initiator; iid=E_S_I_IID; line=18; fact=contains(E_S_I_B,
    secret_Nb_set(E_S_IID)))
% @communication(entity=Initiator; iid=E_S_I_IID; line=20; sender=E_S_I_Actor;
    receiver=E_S_I_B; payload=crypt(pk(E_S_I_B), Nb_1); channel=regularCh; fact=
    iknows(crypt(pk(E_S_I_B), Nb_1)); direction=send)
% @introduce(entity=Initiator; iid=E_S_I_IID; line=20; fact=witness(E_S_I_Actor,
    E_S_I_B, auth_Responder_authenticates_Initiator, Nb_1))
% @step_label(entity=Initiator; iid=E_S_I_IID; line=20; variable=SL; term=5)
step step_004_Initiator__line_18(E_S_IID, E_S_I_Actor, E_S_I_B, E_S_I_IID, Na, Nb
    , Nb_1) :=
  child(E_S_IID, E_S_I_IID).
  iknows(crypt(pk(E_S_I_Actor), pair(Na, pair(Nb_1, E_S_I_B)))).
  state_Initiator(E_S_I_Actor, E_S_I_IID, 3, E_S_I_B, Na, Nb)
  =>
  child(E_S_IID, E_S_I_IID).
  contains(E_S_I_Actor, secret_Nb_set(E_S_IID)).
  contains(E_S_I_B, secret_Nb_set(E_S_IID)).
  iknows(crypt(pk(E_S_I_B), Nb_1)).
  request(E_S_I_Actor, E_S_I_B, auth_Initiator_authenticates_Responder, Na,
      E_S_I_IID).
  request(E_S_I_Actor, E_S_I_B, fresh_Initiator_authenticates_Responder, Na,
      E_S_I_IID).
  secret(Nb_1, secret_Nb, secret_Nb_set(E_S_IID)).
  state_Initiator(E_S_I_Actor, E_S_I_IID, 5, E_S_I_B, Na, Nb_1).
  witness(E_S_I_Actor, E_S_I_B, auth_Responder_authenticates_Initiator, Nb_1)
```

```
% @communication(entity=Responder; iid=E_S_R_IID; line=30; sender=Dummy; receiver
    =E_S_R_Actor; payload=crypt(pk(E_S_R_Actor), pair(E_S_R_Na_1, E_S_R_A_1));
    channel=regularCh; fact=iknows(crypt(pk(E_S_R_Actor), pair(E_S_R_Na_1,
    E_S_R_A_1))); direction=receive)
% @match(entity=Responder; iid=E_S_R_IID; line=30; variable=A; term=E_S_R_A_1)
% @match(entity=Responder; iid=E_S_R_IID; line=30; variable=Na; term=E_S_R_Na_1)
% @fresh(entity=Responder; iid=E_S_R_IID; line=31; variable=Nb; term=E_S_R_Nb_1)
% @introduce(entity=Responder; iid=E_S_R_IID; line=31; fact=secret(E_S_R_Nb_1,
    secret_Nb, secret_Nb_set(E_S_IID)))
% @introduce(entity=Responder; iid=E_S_R_IID; line=31; fact=contains(E_S_R_A_1,
    secret_Nb_set(E_S_IID)))
% @introduce(entity=Responder; iid=E_S_R_IID; line=31; fact=contains(E_S_R_Actor,
     secret_Nb_set(E_S_IID)))
% @communication(entity=Responder; iid=E_S_R_IID; line=32; sender=E_S_R_Actor;
    receiver=E_S_R_A_1; payload=crypt(pk(E_S_R_A_1), pair(E_S_R_Na_1, pair(
    E_S_R_Nb_1, E_S_R_Actor))); channel=regularCh; fact=iknows(crypt(pk(E_S_R_A_1
    ), pair(E_S_R_Na_1, pair(E_S_R_Nb_1, E_S_R_Actor)))); direction=send)
% @introduce(entity=Responder; iid=E_S_R_IID; line=32; fact=witness(E_S_R_Actor,
    E_S_R_A_1, auth_Initiator_authenticates_Responder, E_S_R_Na_1))
% @step_label(entity=Responder; iid=E_S_R_IID; line=32; variable=SL; term=4)
step step_005_Responder__line_30(E_S_IID, E_S_R_A, E_S_R_A_1, E_S_R_Actor,
    E_S_R_IID, E_S_R_Na, E_S_R_Na_1, E_S_R_Nb, E_S_R_Nb_1) :=
  child(E_S_IID, E_S_R_IID).
  iknows(crypt(pk(E_S_R_Actor), pair(E_S_R_Na_1, E_S_R_A_1))).
  not(dishonest(E_S_R_Actor)).
  state_Responder(E_S_R_Actor, E_S_R_IID, 1, E_S_R_A, E_S_R_Na, E_S_R_Nb)
  =[exists E_S_R_Nb_1]=>
  child(E_S_IID, E_S_R_IID).
  contains(E_S_R_A_1, secret_Nb_set(E_S_IID)).
  contains(E_S_R_Actor, secret_Nb_set(E_S_IID)).
  iknows(crypt(pk(E_S_R_A_1), pair(E_S_R_Na_1, pair(E_S_R_Nb_1, E_S_R_Actor)))).
  secret(E_S_R_Nb_1, secret_Nb, secret_Nb_set(E_S_IID)).
  state_Responder(E_S_R_Actor, E_S_R_IID, 4, E_S_R_A_1, E_S_R_Na_1, E_S_R_Nb_1).
  witness(E_S_R_Actor, E_S_R_A_1, auth_Initiator_authenticates_Responder,
      E_S_R_Na_1)

% @communication(entity=Responder; iid=E_S_R_IID; line=33; sender=E_S_R_A;
    receiver=E_S_R_Actor; payload=crypt(pk(E_S_R_Actor), E_S_R_Nb); channel=
    regularCh; fact=iknows(crypt(pk(E_S_R_Actor), E_S_R_Nb)); direction=receive)
% @introduce(entity=Responder; iid=E_S_R_IID; line=33; fact=request(E_S_R_Actor,
    E_S_R_A, auth_Responder_authenticates_Initiator, E_S_R_Nb, E_S_R_IID))
% @introduce(entity=Responder; iid=E_S_R_IID; line=33; fact=request(E_S_R_Actor,
    E_S_R_A, fresh_Responder_authenticates_Initiator, E_S_R_Nb, E_S_R_IID))
% @assignment(entity=Responder; iid=E_S_R_IID; line=34; variable=Na; term=
    E_S_R_Na)
% @introduce(entity=Responder; iid=E_S_R_IID; line=34; fact=secret(E_S_R_Na,
    secret_Na, secret_Na_set(E_S_IID)))
% @introduce(entity=Responder; iid=E_S_R_IID; line=34; fact=contains(E_S_R_A,
    secret_Na_set(E_S_IID)))
% @introduce(entity=Responder; iid=E_S_R_IID; line=34; fact=contains(E_S_R_Actor,
     secret_Na_set(E_S_IID)))
% @step_label(entity=Responder; iid=E_S_R_IID; line=34; variable=SL; term=6)
step step_006_Responder__line_33(E_S_IID, E_S_R_A, E_S_R_Actor, E_S_R_IID,
    E_S_R_Na, E_S_R_Nb) :=
  child(E_S_IID, E_S_R_IID).
  iknows(crypt(pk(E_S_R_Actor), E_S_R_Nb)).
  state_Responder(E_S_R_Actor, E_S_R_IID, 4, E_S_R_A, E_S_R_Na, E_S_R_Nb)
  =>
  child(E_S_IID, E_S_R_IID).
  contains(E_S_R_A, secret_Na_set(E_S_IID)).
  contains(E_S_R_Actor, secret_Na_set(E_S_IID)).
  request(E_S_R_Actor, E_S_R_A, auth_Responder_authenticates_Initiator, E_S_R_Nb,
```

```
      E_S_R_IID).
  request(E_S_R_Actor, E_S_R_A, fresh_Responder_authenticates_Initiator, E_S_R_Nb
      , E_S_R_IID).
  secret(E_S_R_Na, secret_Na, secret_Na_set(E_S_IID)).
  state_Responder(E_S_R_Actor, E_S_R_IID, 6, E_S_R_A, E_S_R_Na, E_S_R_Nb)

section goals:

% @goal(name=auth_Initiator_authenticates_Responder; line=47; AM=AM; AR=AR; AW=AW
    ; IID=E_aIaR_IID)
attack_state auth_Initiator_authenticates_Responder(AM, AR, AW, E_aIaR_IID) :=
  not(dishonest(AW)).
  not(witness(AW, AR, auth_Initiator_authenticates_Responder, AM)).
  request(AR, AW, auth_Initiator_authenticates_Responder, AM, E_aIaR_IID)

% @goal(name=auth_Responder_authenticates_Initiator; line=48; AM=E_aRaI_AM; AR=
    E_aRaI_AR; AW=E_aRaI_AW; IID=E_aRaI_IID)
attack_state auth_Responder_authenticates_Initiator(E_aRaI_AM, E_aRaI_AR,
    E_aRaI_AW, E_aRaI_IID) :=
  not(dishonest(E_aRaI_AW)).
  not(witness(E_aRaI_AW, E_aRaI_AR, auth_Responder_authenticates_Initiator,
      E_aRaI_AM)).
  request(E_aRaI_AR, E_aRaI_AW, auth_Responder_authenticates_Initiator, E_aRaI_AM
      , E_aRaI_IID)

% @goal(name=fresh_Initiator_authenticates_Responder; line=47; FM=FM; FR=FR; FW=
    FW; IID1=IID1; IID2=IID2)
attack_state fresh_Initiator_authenticates_Responder(FM, FR, FW, IID1, IID2) :=
  not(dishonest(FW)).
  request(FR, FW, fresh_Initiator_authenticates_Responder, FM, IID1).
  request(FR, FW, fresh_Initiator_authenticates_Responder, FM, IID2) &
  not(equal(IID1, IID2))

% @goal(name=fresh_Responder_authenticates_Initiator; line=48; FM=E_fRaI_FM; FR=
    E_fRaI_FR; FW=E_fRaI_FW; IID1=E_fRaI_IID1; IID2=E_fRaI_IID2)
attack_state fresh_Responder_authenticates_Initiator(E_fRaI_FM, E_fRaI_FR,
    E_fRaI_FW, E_fRaI_IID1, E_fRaI_IID2) :=
  not(dishonest(E_fRaI_FW)).
  request(E_fRaI_FR, E_fRaI_FW, fresh_Responder_authenticates_Initiator,
      E_fRaI_FM, E_fRaI_IID1).
  request(E_fRaI_FR, E_fRaI_FW, fresh_Responder_authenticates_Initiator,
      E_fRaI_FM, E_fRaI_IID2) &
  not(equal(E_fRaI_IID1, E_fRaI_IID2))

% @goal(name=secret_Na; line=45; Knowers=Knowers; Msg=Msg)
attack_state secret_Na(Knowers, Msg) :=
  iknows(Msg).
  not(contains(i, Knowers)).
  secret(Msg, secret_Na, Knowers)

% @goal(name=secret_Nb; line=46; Knowers=E_sN_Knowers; Msg=E_sN_Msg)
attack_state secret_Nb(E_sN_Knowers, E_sN_Msg) :=
  iknows(E_sN_Msg).
  not(contains(i, E_sN_Knowers)).
  secret(E_sN_Msg, secret_Nb, E_sN_Knowers)
```

References

[1] A. Malhotra et al. XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition). Available at http://www.w3.org/TR/xquery-operators/#regex-syntax, 2007.

[2] M. W. Alford, L. Lamport, and G. P. Mullery. Basic concepts. In M. W. Alford, J.-P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider, editors, *Advanced Course: Distributed Systems*, volume 190 of *Lecture Notes in Computer Science*, pages 7–43. Springer, 1984.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[4] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[5] A. Armando and L. Compagna. SAT-based Model-Checking for Security Protocols Analysis. *International Journal of Information Security*, 7(1):3–32, January 2008.

[6] A. Biere. Bounded model checking. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.

[7] J. Büchi. On a decision method in restricted second order arithmetic. In *Proc. International Congress on Logic, Method, and Philosophy of Science*, pages 1–12. Stanford University Press, 1962.

[8] A. Cimatti, M. Roveri, and D. Sheridan. Bounded verification of past LTL. In A. J. Hu and A. K. Martin, editors, *FMCAD*, volume 3312 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2004.

[9] M. Fisher. A normal form for temporal logics and its applications in theorem-proving and execution. *J. Log. Comput.*, 7(4):429–456, 1997.

[10] D. M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer, 1987.

[11] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal basis of fairness. In P. W. Abrahams, R. J. Lipton, and S. R. Bourne, editors, *POPL*, pages 163–173. ACM Press, 1980.

[12] I. M. Hodkinson and M. Reynolds. Separation - past, present, and future. In S. N. Artëmov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and

J. Woods, editors, *We Will Show Them! (2)*, pages 117–142. College Publications, 2005.

[13] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[14] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[15] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In R. Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[16] N. Markey. Temporal logic with past is exponentially more succinct, concurrency column. *Bulletin of the EATCS*, 79:122–128, 2003.

[17] A. Nerode. Linear automaton transformations. In *AMS*, volume 9. AMS, 1958.

[18] J. Oudinet, A. Calvi, and M. Büchler. Evaluation of aslan mutation operators. In M. Veanes and L. Viganò, editors, *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 178–196. Springer Berlin Heidelberg, 2013.

[19] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[20] A. Prior. *Time and Modality*. Oxford University Press, London, 1957.

[21] A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In M. A. Malcolm and H. R. Strong, editors, *PODC*, pages 39–48. ACM, 1985.

[22] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.

[23] SPaCIoS. Deliverable 2.4.1: Definition of Attacker Behavior Models, 2012.

[24] SPaCIoS. Deliverable 3.2: SPaCIoS Methodology and technology for property-driven security testing, 2013.

[25] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.