MULTI-SESSION SECURITY MONITORING FOR MOBILE CODE

Fabio Massacci and Katsiaryna Naliuka

November 2006

Technical Report # DIT-06-067

Multi-session Security Monitoring for Mobile Code

Fabio Massacci [a] Katsiaryna Naliuka [a]

[a] *University of Trento*

Abstract

There is increasing demand for running multiple times a number of interacting applications in a secure and controllable way on mobile devices. Such demand is not supported by the Java/.NET security models based on trust domains nor by current security monitors or language-based security approaches. Trust domains don't allow for interactions while language-based security doesn't support enough customizable policies.

A careful analysis of the security requirements in the booming domain of mobile games reveals that most practical security requirements can be represented with an enhanced notion of pure past temporal Logic augmented with the intuitive notion of session.

We propose an approach that allows security policies that are i) expressive enough to capture multiple sessions and interacting applications, ii) suitable for efficient monitoring, iii) convenient for a developer to specify them. Since getting all three at once is impossible, we advocate a *logical language*, 2D-LTL a bi-dimensional temporal logic fit for multiple sessions and for which efficient monitoring algorithms can be given, and a *graphical language* based on standard UML sequence diagrams with a tight correspondence between the two.

In this paper we show a refined formal model for capturing the notion of session and the correctness and completeness of the monitoring algorithm for security policies expressed in 2D-LTL.

Email addresses: `Fabio.Massacci@unitn.it` (Fabio Massacci), `naliuka@dit.unitn.it` (Katsiaryna Naliuka).

[1] This work has been partly supported by the EU-IST-STREP-S3MS and FIRB-SECURITY.

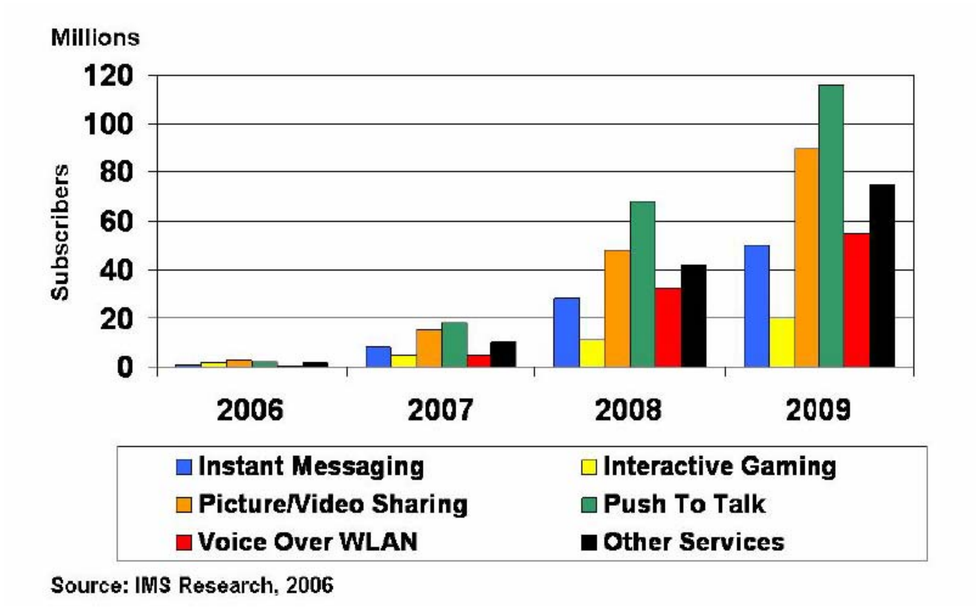


Fig. 1. Expected revenues for the mobile software market

1 Introduction

Mobile devices are increasingly powerful and popular. The smart phone in our pocket has more computing power than the PC encumbering our desk 15 years ago. Yet, if we look at the amount of software available on high-end mobile phones we cannot find even remotely the amount of third party software that was available on our old PC.

One of the reasons for this lack of applications is also the security model adopted for mobile phones. Indeed, the current security model (for instance Java MIDP 2.0) is based on trust relationships: before running an application a device checks that it is signed by a trusted party. Different trust domains segregate applications into sandboxes: inter-operability is either total or not existing. E.g. if a payment service is available on the platform and an application for paying parking meters is loaded in the same trust domain, the user cannot block the parking application from performing large payments.

Most of the current services such as ring-tones, short messages etc. (see Fig. 1) are rather simple and the need for a more sophisticated technology is not apparent.

There is however one notable exception: *Mobile Games*. On-line games are significantly more complex than one can expect [13] and the recent trend of bringing to the mobile massive multi-player online role-playing games, MMMORG for short, introduces additional challenges. Essentially, an MMMORG is a mobile adventure game, which many people can play at once. Distributed

MMMORG keep the role of the game server very small, with most of the game controls run on the mobile device. Such components are able to do complex actions such as silently starting a connection, informing other players by sending them SMS text messages, manipulating the user's address book and so on (see e.g. [21]).

Whereas starting a connection might seem irrelevant in the fixed world of broadband access it can cost a fortune when run on a mobile phone without a business GPRS or UMTS flat rate, or may end up exhausting one's battery just in the moment we need it to make an urgent call. Checking that an application satisfies a number of security properties becomes therefore essential.

Equipping every mobile device with a more flexible security mechanism can be a solution. One of the components of such a system could be a run-time monitor, which controls the program's execution and prevents it from performing illegitimate actions. In this setting, automated reasoning techniques could be extremely useful as they would have a "dual use". Before deployment, security properties could be checked against the mobile game to guarantee the desired level of security before the operator or the developer signs the code. After deployment, we could monitor the satisfaction of the security policies at run-time (we discuss the issue more in details in section 2).

Building such run-time enforcement monitor has been the major goal behind research in security automata [19,4], history-based access control [3,6,12], or usage-based access control [17]. A problem is that the language to express such the policies to be monitored must be [20]

(1) expressive enough to reflect the desired security policies;
(2) suitable for efficient monitoring, as we envisage to enforce it on devices as performance-critical as smart phones;
(3) convenient for a human as we cannot expect a SME developer to be accustomed with logics, type systems or the like.

It is a folk theorem that expressivity, efficiency and usability are hardly satisfied all at once.

However, out of a detailed analysis of the domain [21], it emerges that most user relevant security properties can be essentially captured by a variant of pure-past LTL. Informally, we consider properties of the form "the software will download updates from Internet available only if the user permits it" while we assume that properties like "if the user would like weekly updates, then on sundays the software eventually download all available magic weapons" are taken care of by game's developers.

1.1 Our Contribution

Our proposal is to design *two* languages: a *formal language* that satisfies requirements (1) and (2) and a *graphical language* that satisfies (1) and (3). The formal language guarantees soundness and precision of security policies. The graphical language greatly simplifies specification of policies and security properties. Then we must establish a correspondence between the graphical constructs and the logical policies.

To meet (1+2) we propose a policy language 2D-LTL, based on the past-time linear temporal logic and extended to describe multiple sessions of a program or the concurrent execution of multiple programs. To meet (1+3) we show how 2D-LTL can be used for formalizing UML interaction diagrams [16] exploiting the STAIRS trace semantics for UML [7]. A developer can use these diagrams to specify policies and then translate them in 2D-LTL. We believe the usage of a well know graphical notation is more effective to gauge industry acceptance than inventing yet another graphical notation.

In this, paper we focus on the semantics of the security modelling language and the automated reasoning mechanism needed to build an effective run-time monitor. We provide a natural semantic for multiple sessions and a corresponding 2D-LTL logic (for bi-dimensional LTL). The intuition is that the logic should capture two dimensions in the execution of a program such as a mobile game: the first is the classical dimension of time as a sequence of states traversed by the single run of the game. The second dimention locates the application single run (or session in the intuitive sense of the word) with respect to other applications or to previous runs of the same one.

For sake of clarity, we sketch some idea of the graphical notation, in order to give a grasp of the overall scientific approach, and point the reader to [15] for further details on the translation.

In the rest of the paper we give some motivating examples (§2), introduce our policy language (§3) and discuss the run-time monitoring of policies (§4). Than we provide an overview of UML sequence diagrams(§5) and discuss the formalization of the diagrams in 2D-LTL (§6). Description of the related work(§7) and final remarks end the paper.

2 Motivating Application

When discussing how to secure a mobile game, the issue boils down to the following one: we need a set of methods and tools that decide whether an

Table 1
Security enforcement during application life-cycle

Development	Deployment		Execution
(I) at design and development time	(II) after design but before shipping the application	(III) when downloading the application	(IV) during the execution of the application

action related to a given application is allowed on a mobile platform. Such decisions can be taken at different stages of the application's life-cycle and each stage will have different functionality constraints as detailed in Table 1.

Requirements (I) can be achieved by appropriate design rules and require developer support; (II) and (III) can be carried out through (automatic) verification techniques. Such verifications can take place before downloading (*static analysis and model checking* by developers and operators followed by a contract coming with a *trusted signature*) or as a combination of pre and post-loading operations (e.g., through *in-line monitors* and *proof carrying code*); (IV) can be implemented by *run-time checking.* All methods have different technical and business properties. From an operator's view point:

- working on existing devices would rule out run-time enforcement, and favor static analysis. Monitors may be used (for properties that could not be proved), but on-device proof would then not be possible.
- Operators distrusting the certification process could rely on run-time checks, at the price of upgrading devices' software. In-lining could be used, and contracts could be verified on the device itself.
- An operator who wants to be able to run existing applications would prefer run-time enforcement.

The users' perspectives could be different as individuals might care more of privacy, whereas companies might care more of security. Table 2 shows some of possible strengths and limitations of each technology.

In this scenarios, typical requirements from end users are: [21]:

(1) User should be able to control how much money could be spent by one application during a specific period (at least per session). Accumulated cost and limits should be easily accessible for reading or updating by the user at any time.
(2) User should be able to control uninstalling process of the application, including all the associated files/information stored.
(3) Each time the application accesses specific device/user information (not referred to its own information), user should be able to control the access.
(4) Any relevant information sent by the application outside of the device should be controlled by the user.

Table 2
Technologies Strengths and Weaknesses

Criteria	Static Analysis	In-lining	Run-time
Works with existing devices	$\surd$	?	$\times$
Works with existing applications	?	$\times$	$\surd$
Does not modify applications	$\surd$	$\times$	$\surd$
Offline proof of correctness	$\surd$	$\surd$	$\times$
Load-time proof of correctness	$\times$	$\surd$	$\times$
May depend on run-time data	$\times$	$\surd$	$\surd$
Does not affect runtime performance	$\surd$	?	$\times$

(5) While the application is running some accesses to other applications could be required. For this reason, user should be notified and should be able to control an access.
(6) The application should not run if the battery is below a certain value specified by the user.

By transforming such user requirements into security properties we end up with properties of the following form:

- *permitting or prohibiting the activation or deactivation of a security relevant service* (e.g. opening a communication, sending an SMS text, starting an application, modifying the address book etc.)
- *presence of past events as a pre-requisite for allowing another present event* (e.g. the user confirmation before an SMS is send or an image is downloaded)
- *cumulative accounting of events* (e.g. the application only loads each image from the network once)
- *enabling or disabling features since an initial event took place*, for instance disabling the game's silent calls since the battery fell down a certain level.

However, when stating more precisely the above requirements, an additional notion comes into play: the notion of *session*. This notion is fairly clear from a user's point of view: we started an application, we run it and then we terminated it. End of the session.

The constraints on what may happen during a session can be easily characterized with pure-past linear temporal logic: "you can do A only if *previously* you did B"; "you can do C if you have been doing B *since* you did A" and so on. Time is linear and rooted, it starts at the moment we invoke the application and ends when we terminate it.

Multiple sessions are also intuitively clear: we start an application, we run, we terminate it; then we restart again, we run it again, we stop it; then we restart it, and so on. Each time it is a new session. In some cases it is the same application that it is re-started again, in other cases there might be other applications.

Let's see two examples:

Example 1 *A mobile phone user participates in a multi-player online game. Communication during the game is performed through MMS messages. The user's mobile operator provides a number N of MMS messages per month for free. So, the user wants the game applet to send no more than N messages every month. When this limit is reached, the applet must suspend the game and ask the user whether he is willing to continue playing. If the answer is yes, the game must be resumed, else it must be terminated.*

Example 2 *In Example 1 the user can answer* "no to all". *Then the applet must terminate its execution without further ado each time the limit is reached in subsequent executions.*

The two examples below shows that we can aggregate these constraints into two large classes:

constraints on the current session that describe the security behavior of the application from the moment in which it is invoked to the moment in which it is turned off.

constraints across multiple sessions of the application (or its interaction with applications that started before actually invoking the current application)

Both properties can be expressed in terms of past behavior of actions and objects' properties belonging to the application with different temporal operators.

Another important observation is that all such constraints are always characterized by what we might call the *monitoring assumption*: they must be true at any single moment during the execution of the game.

3 A bi-dimensional model of execution

As we already said, we can consider a model of the system as a bi-dimensional model with two "time" dimensions.

The horizontal time dimension determines the progressing of an application during a run i.e. the sequence of events from the moment it is activated to the moment it terminates. The vertical dimension represents the active sessions that have been started by the system and not terminated. We can imagine that the execution starts on the ground floor, and the progresses on the same floor sweeping more and more rooms. At a certain point we start a new session (of another application or another instance of the old one), climb to the new floor and keep moving in that floor. Occasionally we remember that we have left our previous session half-ways, return to the those previous floor and keep sweeping there. Than we will keep progressing by either spawning new floors or sweeping new rooms.

The existence of these two levels of execution was underlined by Abadi and Fournet in [1] where they are called *sensitive access requests history* (sweeping room after room) and *history of control transfers* (climbing from floor to floor).

To define the model we start with some preliminary definitions.

Definition 1 *Let E be a set, a* trace *t over a E is a sequence of elements from E. The set of all possible non empty traces over E is denoted by E^+. If $t \in E^+$ we will denote the length of the trace t as $|t|$ and the i-th element as $t[i]$. The following operations on traces are defined:*

concatenation $t_1 \circ t_2$ *gluing two traces together.*
truncation $t[..i]$ *taking the prefix of length i of trace t*
filtering $E' \text{Ⓢ}\ t$ *$(E' \subseteq E)$ the result of this operation is another trace t' that is obtained from t by removing all elements that are not contained in A.*

For example $\{e_1, e_2\} \text{Ⓢ} \langle e_2, e_3, e_2, e_1\rangle = \langle e_2, e_2, e_1 *\rangle$.

3.1 A traditional linear model

The traditional model of execution is a sequence of events, such as starting or ending a process, or updating a state of the running process when it has performed new actions. Besides events, we define an additional set of labels L, which contains unique labels for all processes (i.e. the pair application and session) run on the device. So every session of the application is associated with exactly one label $l \in L$ [2].

Definition 2 *Let L be a set of labels and P a set of boolean predicates that represents possible actions a process can perform. A* serialized execution *t is a sequence over the events $E = \{new(l), update(P', l), end(l)\}$ where $P' \subseteq P$ (actions that have occurred in the updated process) and $l \in L$ is subject to the following constraints:*

[2] In practice, this can be easily done with signed assemblies in .NET. and similar features in Java.

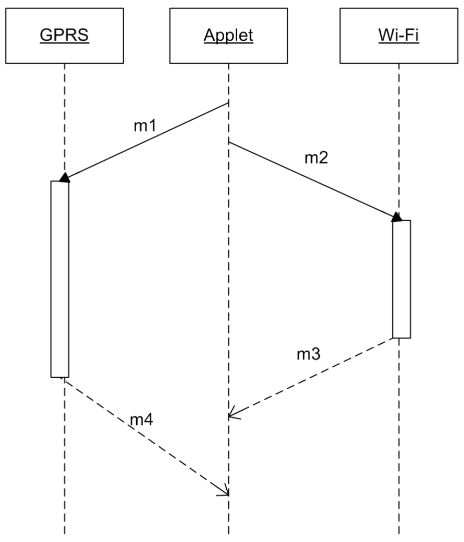
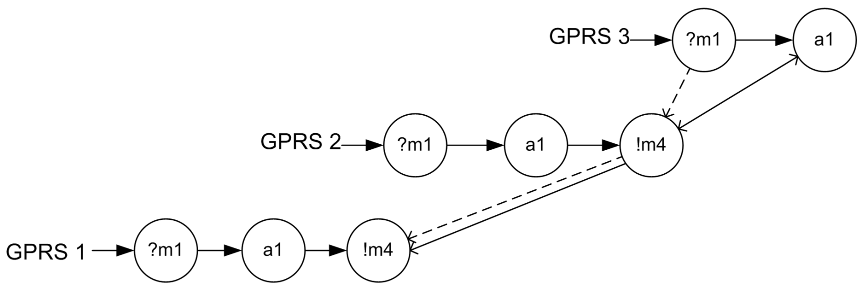


Fig. 2. Interaction between an applet and GPRS/Wifi transmitters (Example 3)



$\langle new(1), update(\{?m_1\}, 1), update(\{a_1\}, 1), update(\{!m_4\}, 1), end(1)\rangle \circ$

$\langle new(2), update(\{?m_1\}, 2), update(\{a_1\}, 2), update(\{!m_4\}, 2), end(2)\rangle \circ$

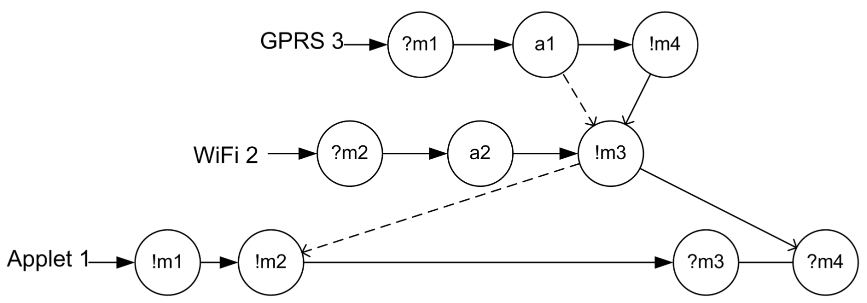
$\langle new(3), update(\{?m_1\}, 3), update(\{a_1\}, 3)\rangle$

Fig. 3. Non-resuming sessions (Example 3)

(1) $new(l)$ *occurs at most once,*
(2) $end(l)$ *occurs at most once,*
(3) if $update(P', l)$ *occurs in the trace then* $new(l)$ *occurs before it in the trace and* $end(l)$ *does not occur before it,*
(4) if $end(l)$ *occurs in the trace then* $new(l)$ *occurs before it.*

Example 3 *A mobile device is equipped with GPRS and Wi-Fi transmitters, and the applet must check the availability of either network. It sends two messages* m_1, m_2 *to the network interfaces (actually these are invocations of APIs), and the transmitters perform the checks* a_1 *and* a_2*. Than transmitters report back the states of each network by messages* m_3 *and* m_4*. Figure 2 illustrates the behavior of the system by the diagram.*

We denote by $!m$ *a predicate "sending a message* m*". Receiving a message we*



$\langle new(1), update(\{!m_1\},1), update(\{!m_2\},1), new(2), update(\{?m_2\},2)\rangle \circ$
$\langle new(3), update(\{?m_1\},3), update(\{a_2\},2), update(\{a_1\},3)\rangle \circ$
$\langle update(\{!m_3\},2), end(2), update(\{!m_4\},3), end(3), update(\{?m_3\},1), update(\{?m_4\},1)\rangle$

Fig. 4. Resuming sessions (Example 3)

will denote as $?m$.

Consider the situation from the point of view of GPRS device.: session starts when the device is invoked by a message m_1 from the applet; than it performs a check a_1; reports the state of the network by another message m_3. Than a session terminates, and if the applet is invoked once again a new session starts. The serialized trace of the execution from this point of view is depicted at Figure 3.

On the other hand, if we consider evolution of the system as a whole, the session of the main applet is never terminated but only suspended when GPRS and Wi-Fi transmitters perform their tasks. Figure 4 presents a trace of execution for the whole system.

Such operational representation is a faithful reflection of what happens in reality, with single threaded execution of "concurrent" applets or repeated sessions of the same applet.

To prove the correctness of the monitoring algorithm we need a series of additional definitions:

Definition 3 *Let t be a serialized execution over L and P. Given a label l such that $new(l)$ occurs in t, the label of the previous session with respect to l is the label $prec\,(l,t)$ such that $new(prec\,(l,t))$ occurs before $new(l)$ in t and there is not other $new(l')$ between them in t.*

Unfortunately, serialized traces are suitable only for expressing global properties in LTL that disregard completely the structure of sessions. Expressing temporal properties of sessions in this model turned out to be significantly convolute and does not correspond to the intuitions that users have about the system.

Technically, we link states of each application by using two threads: the *session* and the *frontier*. A session represents the sequence of states corresponding to a single execution of an application. A frontier is formed of the last active states of all previously started sessions. From an application perspective, the session represents what it did by running itself. The frontier is the point of arrival of what the others did so far. The frontier formed by the last events of all sessions is the *current frontier* (see Fig. 4).

Definition 4 *A* history *is a tuple $H = \langle S, s_f, \mathcal{T}, \mathcal{F}, L, P, V\rangle$, where S is a set of* states, *a special state $s_f \in S$, is the final state of the history, the functions $\mathcal{T}, \mathcal{F} : S \to S^+$ link every state to a sequence of states, i.e. its session and its frontier, $V : P \to 2^S$ is an assignment of predicates from a set P to a set of states. L is a set of labels of sessions and the following conditions hold:*

Session past determinism *For all $s \in S$ if $\mathcal{T}(s) = t \circ \langle s_{-1}, s\rangle$ then $\mathcal{T}(s_{-1}) = t \circ \langle s_{-1}\rangle$*

Current frontier past determinism *If $\mathcal{F}(s_f) = f_1 \circ \langle s'\rangle \circ f_2$ then $\mathcal{F}(s') = t_1 \circ \langle s'\rangle$*

Mutual past determinism *For all $s \in S$ if $\mathcal{F}(s) = f \quad \cdot s' \cdot s$, $\mathcal{T}(s) = t \circ \langle s_{-1}, s\rangle$ exists $s'' \in \mathcal{T}(s')$ such that $\mathcal{F}(s_{-1}) = f' \circ \langle s'', s_{-1}\rangle$*

Intuitively s_f corresponds to the final state of the last open session. $\mathcal{T}(s)$ returns a prefix of the session, to which s belongs, up to s itself inclusive. Similarly, $\mathcal{F}(s)$ is a frontier formed of the states of all previously started sessions. The frontier $\mathcal{F}(s_f)$ is a current frontier and includes final states of all session. If s is the final state of its session $\mathcal{F}(s)$ is a prefix of $\mathcal{F}(s_f)$ (final past determinism). When a new event occurs after s in its session and s is no longer the final state $\mathcal{F}(s)$ becomes frozen and does not change any more.

To clarify the concept of sessions and frontiers let us return to Example 3. For the GPRS applet manager execution is a sequence of successive sessions where new session cannot start until the old one has completely finished its execution. Then a frontier for state s includes final states of all preceding sessions and s itself. At Figure 3 a solid line depicts the frontier for state $!m_1$ of Session 3 (the current frontier) and a dashed line shows the frontier for the state a_1 of the same session. It can be seen that they coincide in all points but the last.

From the perspective of the system as a whole the previously suspended sessions can be resumed again. The example of how our model reflects this situation is depicted at Figure 4. The frontier for final states of all sessions succeeding the resumed one will change. For instance, the frontier for $!m_4$, depicted in the figure by the solid curve, includes state $?m_4$ of Session 1, even

$$
\begin{array}{lll}
t\,[..i]\,, f\,[..j] \vDash p, p \in P & \text{iff} & p \text{ holds in } (t\,[..i]\,, f\,[..j]) \\
t\,[..i]\,, f\,[..j] \vDash \neg\psi & \text{iff} & t\,[..i]\,, f\,[..j] \nvDash \psi \\
t\,[..i]\,, f\,[..j] \vDash \psi_0 \;\vee\; \psi_1 & \text{iff} & t\,[..i]\,, f\,[..j] \vDash \psi_0 \text{ or } h_i[..j] \vDash \psi_1 \\
t\,[..i]\,, f\,[..j] \vDash Y_G\;\psi & \text{iff} & j > 1 \text{ and } \mathcal{T}\,(f\,[j-1])\,, f\,[..j-1] \vDash \psi \\
t\,[..i]\,, f\,[..j] \vDash \psi_0\;S_G\;\psi_1 & \text{iff} & t\,[..i]\,, f\,[..j] \vDash \psi_1 \text{ or } j > 1 \text{ and} \\
 & & \mathcal{T}\,(f\,[j-1])\,, f\,[..j-1] \vDash \psi_0\;S_G\;\psi_1 \text{ and} \\
 & & t\,[..i]\,, f\,[..j] \vDash \psi_0 \\
t\,[..i]\,, f\,[..j] \vDash Y_L\;\psi & \text{iff} & i > 1 \text{ and } t\,[..i-1]\,, \mathcal{F}\,(t\,[i-1]) \vDash \psi \\
t\,[..i]\,, f\,[..j] \vDash \psi_0\;S_L\;\psi_1 & \text{iff} & t\,[..i]\,, f\,[..j] \vDash \psi_1 \text{ or } i > 1 \text{ and} \\
 & & t\,[..i-1]\,, \mathcal{F}\,(t\,[i-1]) \vDash \psi_0\;S_L\;\psi_1 \text{ and} \\
 & & t\,[..i]\,, f\,[..j] \vDash \psi_0
\end{array}
$$

Fig. 5. 2D-LTL Recursive semantics

though the corresponding event happened later in the execution. However, a frontier for a_1 (depicted by the dashed line) do not change since this event is no longer final in its session. It belongs to the past of Session 3 and its frontier cannot be changed any more.

For our own policy language 2D-LTL we use the following operators: $\neg$, $\vee$, Y_L ("in the previous state of this session"), S_L ("since in this session"), Y_G ("in the previous session"), S_G ("since between sessions"). So if $p \in P$ are atomic propositions, then 2D-LTL formulae are:

$$F ::= \bot \mid \top \mid p \mid \neg F \mid F_1 \vee F_2 \mid Y_L\;F \mid F_1\;S_L\;F_2 \mid Y_G\;F \mid F_1\;S_G\;F_2$$

Definition 5 *A history $H = \langle S, s_f, \mathcal{T}, \mathcal{F}, P, V\rangle$ satisfies 2D-LTL formula ϕ ($H \models \phi$) iff $\mathcal{T}\,(s_f)\,, \mathcal{F}\,(s_f) \models \phi$ where $\models$ is evaluated in a recursive way along the rules of Figure 5.*

Other local (respectively global) operators can be expressed by a combination of the primitive local (resp. global) operators. For example:

in some session in the past $O_G\;\psi = true\;S_G\;\psi$
in some moment in the past in this session $O_L\;\psi = true\;S_L\;\psi$
historically in every session in the past $H_G\;\psi = \neg O_G\;\neg\psi$
historically in the past in this session $H_L\;\psi = \neg O_L\;\neg\psi$

If we limit state predicates to statements about occurrence of events in the session and do not use local operators, we obtain the model[3] by Krukow et al.[12]. If we only have one session, the model is equivalent to Havelund and Roşu's [9]. We discuss this issue more in detail in the Section 7.

We do not make any assumption on the values of the predicates used in the formula. It is possible to use any computable predicates on the state of execution.

To make the examples of how our policy language can be used, let us produce the formulae that encode the policies of our running example. For writing all policies, we use the predicate $MMS\_limit\_achieved(s)$ and the predicates *yes_sent*, *no_sent*, and *no_to_all_sent* that evaluate to true since the message was sent until it was received. Also we use the events *suspend*, *resume terminate*, *ask_user*, *answer_yes*, *answer_no*, and *answer_no_to_all*, and the application label *game*.

Example 4 *If the MMS messages limit was achieved, suspend the execution.*

$$H_G\ (Y_L\ (MMS\_limit\_achieved) \rightarrow suspend)$$

We use H_G to ensure that the policy hold in every session (that means that in our game applet too).

Example 5 *After the MMS messages limit was achieved and the game was suspended, the user must be asked whether he wants to continue playing.*

$$H_G\ (Y_L\ (Y_L\ (MMS\_limit\_achieved) \wedge suspend) \rightarrow ask\_user)$$

For the complicated version of our example we should also set a policy, which ensures the game to be terminated without asking, if the user has answered "*no_to_all*" in some previous session of the game.

Example 6 *After the MMS messages limit was achieved and the game was suspended, if the user answered "no to all" in a previous session of the game, the applet must be terminated.*

$$\begin{aligned} H_G\ (\ (Y_L\ (suspend \wedge Y_L\ MMS\_limit\_achieved) \wedge \\ O_G\ (game \wedge O_L\ answer\_no\_to\_all)\ ) \rightarrow terminate) \end{aligned}$$

The next example is taken from [12].

[3] We do not consider their "possible" operator, which they also ignore in all their examples.

Example 7 *eBay transaction after closing the auction consists of the following steps: the winning bidder sends payment for the bid; finally the seller sends the bid to the winner. Moreover, at every stage positive, negative, or neutral feedback can be provided by any side (after giving feedback it can no longer be changed). We use the predicates pay (buyer sends payment), pos, neg, neu when the seller provides a positive, negative, or neutral feedback.*

When a customer decides whether he wants to bid at someone's auction he may use the following criteria among others: "The seller has never provided negative feedback after the payment is made". Here the order of the events is crucial. This policy can be expressed in 2D-LTL in the following way:

$$H_G\ \neg O_L\ \ (neg \wedge O_L\ pay)$$

In comparison with [12] we can capture this policy without introducing a special event "ignore" (unobservable for everyone but the bidder himself).

Definition 6 *A* corresponding history $H = \mathcal{H}(t)$ *for serialized trace of execution t is constructed recursively by the following rules:*

(1) To a trace consisting of the event $t = \langle new(l) \rangle$ *corresponds a history* $H = \left\langle \{s_0^{(l_1)}\}, s_{l_1}^0, \mathcal{T}, \mathcal{F}, \{l_1\}, \emptyset, V \right\rangle$, *where* $\mathcal{T}\left(s_0^{(l_1)}\right) = \left\langle s_{l_1}^0 \right\rangle$, $\mathcal{F}\left(s_0^{(l_1)}\right) = \left\langle s_0^{(l_1)} \right\rangle$. *It is obvious that H satisfies the properties of the history.*

(2) With every new event in the trace the system evolves from H to H′ according to its type:

new(l)

$$S' = S \cup \{s_0^{(l)}\}, s_0^{(l)} \notin S$$

$$\mathcal{T}'(s) = \begin{cases} \left\langle s_0^{(l)} \right\rangle, \text{ if } s = s_0^{(l)} \\ \mathcal{T}(s) \quad otherwise \end{cases}$$

$$\mathcal{F}'(s) = \begin{cases} \mathcal{F}(s_f) \ \circ \left\langle s_0^{(l)} \right\rangle, \text{ if } s = s_0^{(l)} \\ \mathcal{F}(s) \qquad\qquad otherwise \end{cases}$$

$$L' = L \qquad P' = P$$

$$V'(p) = V(p) \quad s'_f = s_0^{(l)}$$

update(P_{new},l_k) *Let* $\mathcal{F}(s_f) = \left\langle s_{i_1}^{(l_1)}, \ldots, s_{i_{k-1}}^{(l_{k-1})} s_i^{(l_k)}, \ldots, s_{i_{n-1}}^{(l_{n-1})}, s_f \right\rangle$, *label of the last open session is* l_n.

$$S' = S \cup \{s_{i+1}^{(l_k)}\},\ \textit{where } s_{i+1}^{(l_k)} \notin S$$

$$\mathcal{T}'(s) = \begin{cases} \mathcal{T}\left(s_i^{(l_k)}\right) \circ \left\langle s_{i+1}^{(l_k)} \right\rangle, & \textit{if } s = s_{i+1}^{(l_k)} \\ \mathcal{T}(s) & \textit{otherwise} \end{cases}$$

$$\mathcal{F}'(s) = \begin{cases} \mathcal{F}\left(s_{i_{k-1}}^{(l_{k-1})}\right) \circ \left\langle s_i^{(l_k)}, \ldots, s_{i_j}^{(l_j)} \right\rangle, & \textit{if } s = s_{i_j}^{(l_j)}, k \le j \le n \\ \mathcal{F}(s) & \textit{otherwise} \end{cases}$$

$$L' = L \qquad\qquad P' = P \cup P_{new}$$

$$V'(p) = \begin{cases} V(p) \cup \{s_{i+1}^{(l_k)}\}, & \textit{if } p \in P_{new} \\ V(p) & \textit{otherwise} \end{cases} \qquad s'_f = \begin{cases} s_{i+1}^{(l_k)}, & \textit{if } l_k = l_n \\ s_f & \textit{otherwise} \end{cases}$$

end(l_k) $H' = H$ *(nothing changes in the history).*

Proposition 1 *If $\mathcal{H}(t)$ is a valid history then $\mathcal{H}(t \circ \langle e \rangle)$is also a history.*

For the proof of Proposition 1 see Appendix A.

4 Run-time verification

We outline how the monitoring process can be organized. Since sessions in the past can be reactivated (when the controls return to them) it is impossible to evaluate the formula recursively by looking only one step backward [10,12]. So we store the last state of each application since the first not terminated one. When a new event is captured, the evaluation starts from the session where this event happened and proceeds through all the applications that have started after that session. Evaluation of each session uses the values obtained from the previous computation.

As values of elements do not depend only on the values of succeeding elements in the same session, nor on the values in succeeding session, a bi-dimensional recursive calculation can be derived by using the recursive semantics from Section 3 . We show the intuitions behind the algorithm in Fig.6.

Essentially the intuition is that for all currently active sessions i from 1 to l_f the following properties holds:

- $\phi_{preloc}[i]$ contains the evaluation of all subformulae of 2D-LTL formula ϕ in state $t\,[..i_j - 1]\,, \mathcal{F}\,(t\,[..i_j - 1])$. Obviously at the beginning of the trace we set it to *nil*,
- pϕ_{pregl}^{j} contains the evaluation of the formula in state $\mathcal{T}\,(s_{j-1})\,, \mathcal{F}\,(s_{j-1})$. If $j = 1$ then it is also set to *nil*.

```
Algorithm AbstractMonitor;
input a 2D-LTL formula $\phi$;
} a sequence of events from the target $E$;
throws SecurityException if event $e \in E$ violates $F$
variable integer $l_f$;
begin
  $l_f$=0;
  while $E$ not empty do
        pick $e$ from $E$;
        if $e$.type == new then
              $l_f = l_f$+1;
              create $\phi^{l_f}_{now}$, $\phi^{l_f}_{pregl}$, $\phi^{l_f}_{preloc}$ from $\phi$;
              Evaluate($\emptyset$, $\phi^{l_f}_{now}$, $\phi^{l_f}_{pregl}$, $nil$);
        else if $e$.type == update(P, l) then
              $\phi^{l}_{preloc} = \phi^{l}_{now}$, $\phi^{l}_{pregl} = \phi^{l-1}_{now}$;
              Evaluate(P, $\phi^{l}_{now}$,$\phi^{l}_{pregl}$, $\phi^{l}_{preloc}$);
              for j=l+1 to $l_f$ do
                    Evaluate($nil$, $\phi^{j}_{now}$, $\phi^{j}_{pregl}$, $\phi^{j}_{preloc}$);
        if not $\phi^{l_f}_{now}$ then throw SecurityException;
end
```

Fig. 6. The simplified algorithm for monitoring the target application

- P is a set of predicates true in s_j (which can be the empty set $\emptyset$, or $P = nil$.
- after executing **Evaluate** $\phi^{l_f}_{now}$ contains the evaluation of the formula in state $\mathcal{T}(s_j), \mathcal{F}(s_j)$.

The detailed monitoring algorithm is presented in Fig. 8 and Fig. 7.

At first we must detail the construction of the **Evaluate** function assuming its input arrays contain the right information. So, we break down the formula ϕ into sub-formulae in such a way that we can recursively evaluate it as one by Havelund and Roşu [10] and Krukow [12]. Sub-formulae are placed in such order that if now[k_1] is a sub-formula of now[k_2] then $k_1 > k_2$. This constructions allows one to evaluate quickly a formula on the basis of the pre-computed value of the sub-formulae.

For the correctness of the **Evaluate** function we will simply make use of the following facts:

- the pre_loc[0..m] array contains the evaluation of all subformulae of 2D-LTL formula in state $t[..i_j - 1], \mathcal{F}(t[..i_j - 1])$. If $i_j = 1$ then pre_loc[0..m]= nil,
- pre_gl[0..m] contains the evaluation of the formula in state $\mathcal{T}(s_{j-1}), \mathcal{F}(s_{j-1})$. If $j = 1$ then pre_gl[0..m]= nil,
- identical conditions for the set of predicates P as we have shown in the AbstractMonitor.

```
Evaluate values of the subformulae
input
   a set of predicates $P$
         //(if nil then the predicates are unchanged with respect to now)
   an ordered array of subformulae now
   an ordered array of subformulae pre_gl
         //(if nil this is the first session)
} an ordered array of subformulae pre_loc
         //(if nil this the session has just started)
output
   now is modified and contains updated values for all subformulae
begin
   for k=now.legth-1 downto 0 do
         case now[k].type of
         predicate $p$:
               if $P = nil$ then skip; //value unchanged
               else if $p \in P$ then now[k]=true
               else now[k]=false;
         $\neg\phi_i$:
               now[k]=not now[i];
         $\phi_i \wedge \phi_j$:
               now[k]=now[i] and now[j];
         $\phi_i \vee \phi_j$:
               now[k]=now[i] or now[j];
         $Y_L\ \phi_i$:
               if pre_loc[i] == nil then now[k]=false;
               else now[k]=pre_loc[i];
         $Y_G\ \phi_i$:
               if pre_gl[i] == nil then now[k]=false;
               else now[k]=pre_gl[i];
         $\phi_i\ S_L\ \phi_j$:
               if pre_loc[i] == nil then now[k]=now[j];
               else now[k]=now[j] or now[i] and pre_loc[k];
         $\phi_i\ S_G\ \phi_j$:
               if pre_gl[i] == nil then now[k]=now[j];
               else now[k]=now[j] or now[i] and pre_gl[i];
end
```

Fig. 7. The algorithm for recursive evaluation of 2D-LTL formulae

Then it is easy to prove by induction the after executing the function **Evaluate** the now[0..m] array contains the evaluation of the formula in state $\mathcal{T}(s_j), \mathcal{F}(s_j)$.

Now we have the machinery for full monitoring algorithm Fig. 8.

```
Algorithm Monitor;
input a 2D-LTL formula $\phi$;
  a sequence of events from the target $E$;
throws SecurityException if event $e \in E$ violates $F$
variable an array of subformulae $\mathcal{F}$;
  integer $l_f$;
begin
  $l_f$=0;
  now[0]=$nil$;
  $\mathcal{F}$=decompose($\phi$);
  order $\mathcal{F}$ by inclusion so that if $\mathcal{F}[i]$ is a subformula of $\mathcal{F}[j]$ then $i > j$;
  while $E$ not empty do
      pick $e$ from $E$;
      if $e$.type == new then
          $l_f = l_f$+1;
          create arrays now[$l_f$],pre[$l_f$] from $\mathcal{F}$;
          Evaluate($\emptyset$, now[$l_f$], now[$l_f$-1], $nil$);
      else if $e$.type == update(P, l) then
          pre[l]=now[l];
          Evaluate(P, now[l], now[l-1], pre[l]);
          for j=l+1 to $l_f$ do
              Evaluate($nil$, now[j], now[j-1], pre[j]);
      if not now[$l_f$][0] then throw SecurityException;
end
```

Fig. 8. The algorithm for monitoring

We can now state the main result of the paper:

Theorem 1 *Let t be a serialized execution over E and let ϕ be a 2D-LTL formula. Then now*$[l_f][0] == 1$ *iff* $\mathcal{H}(t) \models \phi$.

The proof is done by induction on the number of elements in the monitored trace. We show that a strong invariant that links the value of the *now* arrays to the value of the monitored formula at key points in the history corresponding to the serialized trace (for details of the proof see Appendix B).

In a nutshell, if the number of sessions in the execution is equal to l_f and $\mathcal{F}(s_f) = \langle s_1, \ldots, s_l \rangle$ is the current frontier of execution then for all $j = 1..l_f$ the following properties hold:

(1) now[j] contains the result of evaluation of the formula in state s_j,
(2) if $\mathcal{T}(s_j) = t[..i_j]$ then pre[j] contains the result of evaluation of the formula in state $s_j^{(-1)} = t[i_j - 1]$,
(3) now$[l_f][0..n]$ is evaluated in s_f.

```
**Algorithm** OptMonitor;
**input** a 2D-LTL formula $\phi$;
    a sequence of events from the target $E$;
**throws** SecurityException if event $e \in E$ violates $F$
**variable** an array of subformulae $\mathcal{F}$;
    integer $l_f$;
    a heap of integers RunningSessions;
            //*top element contains minimal element* **min**(RunningSessions);
**begin**
    $l_f$=0;
    RunningSessions=$\emptyset$;
    now[0]=*nil*;
    $\mathcal{F}$=**decompose**($\phi$);
    **order** $\mathcal{F}$ **by** inclusion so that if $\mathcal{F}[i]$ is a subformula of $\mathcal{F}[j]$ then $i > j$;
    **while** $E$ not empty **do**
        **pick** $e$ **from** $E$;
        **if** $e$.type == new **then**
            $l_f = l_f$+1;
            **add** $l_f$ **to** RunningSessions;
            //*as in* Monitor
        **else if** $e$.type == update(P,l) **then**
            pre[l]=now[l];
            **Evaluate**(P, now[l], now[l-1], pre[l]);
            **for** j=l+1 **to** $l_f$ **do**
                **Evaluate**(*nil*, now[j], now[j-1], pre[j]);
        **else if** $e$.type == end(l) **then**
            **remove** l **from** RunningSessions;
            **destroy** pre[l];
            **for** j=max(2, l) to **min**(RunningSessions)-1 **do**
                **destroy**(new[j-1])
        **if not** now[$l_f$][0] **then throw** SecurityException;
**end**
```

Fig. 9. The optimized algorithm for monitoring

More details are provided in the appendix.

Certain optimization of this algorithm can be performed. In particular, there is no need to store values of the arrays for the i-th session if all sessions $1, \ldots, i$ are terminated (and so cannot receive new upcoming events) [12]. However, we must store i if some $j < i$ is still active because the value of global operators may change on i. The corresponding algorithm is shown in Figure9.

The key idea is that the algorithm will only have to use

- pre[l], $l \in$*RunningSessions*,

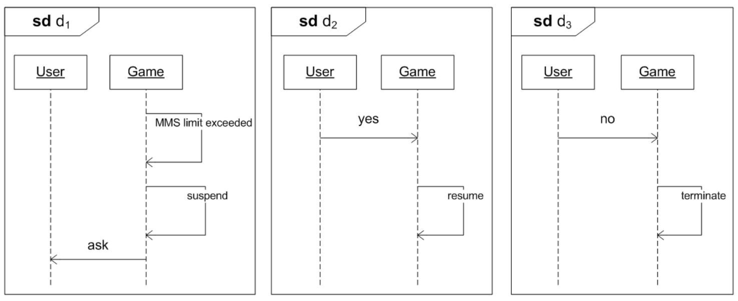


Fig. 10. Diagrams representing the desired behavior of the game applet

- now[l], for all sessions in ***min**(RunningSessions)*$-1 < l \leq l_f$

Then for the formal proof of correctness of the monitoring algorithm we simply make the assumption that destroyed vectors continue to hold the values they stored at the time of destruction but simply cannot be accessed by the algorithm (for details see Appendix C).

5 Overview of UML sequence diagrams

UML sequence diagrams are used to represent interactions between applications. Each session of the application is represented at the diagram as a lifeline (vertical line) with a sequence of events on it. Events represent transmitting and receiving the messages. A message is a triple $m = \langle s, tr, re \rangle$, where s is a unique name of the message, and tr, re are correspondingly transmitting and receiver events. The transmitter of a message m is denoted as $!m$, and the receiver as $?m$. The events on each lifeline are ordered. Nothing can be said about ordering of the events from different lifelines except when they are a transmitter and a receiver of the same message. Then the transmitter must necessarily precede the receiver. For more information about sequence diagrams see [16].

Now let us construct sequence diagrams representing our simple example scenario. It includes two acceptable variants of execution: when the user decides to continue playing and the program resumes, and when he prefers the program to be terminated. Both these scenarios have one part in common – after the limit of MMS is exceeded, the applet must suspend execution and ask a user, whether he wants to continue the game. Therefore, we represent the example scenario by three diagrams, as can be seen at Figure 10. Diagram d_1 represents the common part, while d_2 and d_2 represent differing variants of execution.

These diagrams are combined into a single whole by application by UML op-

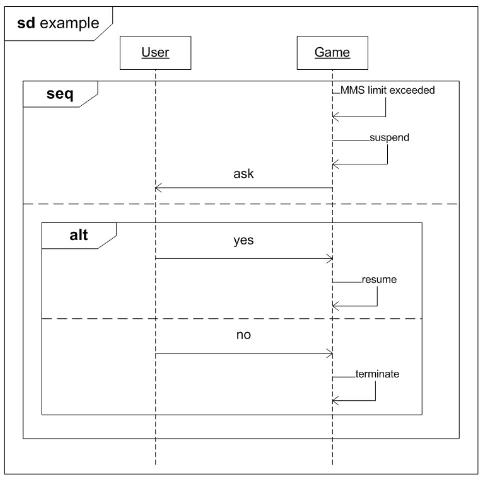


Fig. 11. The combined diagram representing the desired behavior of the game applet

erators. UML operators can specify both interconnections between diagrams (*seq*, *strict*, *alt*) and the meaning of the diagram in the description of the interaction (as *refuse*). In this paper we consider the following UML 2.0 operators:

seq[<**list of interactions**>] the weak sequencing operator combines the operands by preserving ordering of events on each lifeline with respect to the ordering of the operands. Does not put any constraints on the ordering of events from different lifelines.

alt[<**list of interactions**>] All enumerated variants of behavior are acceptable (possibly, depending on some constraints).

refuse[<**interaction**>][4] The behavior is unacceptable.

The three diagrams from Figure 10 can be merged in one as shown in Figure 11: $d = seq\,[d_1, alt[d_2, d_3]]$. It means that after the first part took place, it is obligatory for a program to follow one of the two branches.

Further in the paper we assume, that every message and every event is unique both throughout the diagram and the execution, and not pay attention to checking it (though such properties can be modelled in 2D-LTL).

6 From UML interaction diagrams to 2D-LTL

The goal of this section is to demonstrate how a security policy in 2D-LTL can be defined for every UML interaction. We only describe here the translation

[4] For the reason, why we use *refuse* operator, and not *neg*, see [8].

of interaction diagrams because state diagrams can be easily translated into finite state automata.

In our interpretation of UML sequence diagrams we rely on the STAIRS semantic [7]. In this semantic UML interaction diagrams imply two kinds of constraints on the execution: those defining the acceptable (positive) traces of execution, and those defining the unacceptable (negative) ones. Some traces cannot be put in any category and are called inconclusive. A simple message diagram can only specify positive traces; negative traces appear as a result of applying UML operators to the diagram (e.g the *refuse* operator).

Let us establish a correspondence between STAIRS traces and our bi-dimensional histories. First we must create a link between *states* of our histories and *events* that form traces. For this reason we define the set of states S as $E \times L$, the predicates $e_1, \ldots, e_n$ corresponding to all the events in the system and $l_1, \ldots, l_n$ corresponding to the lifelines.

To encode every UML interaction (either a single diagram or a composition of diagrams) we will use seven 2D-LTL formulae. Three of them represent respectively the precondition, the postcondition, and the invariant of the positive constraints, and three others in a same way define the negative constraints. An auxiliary synchronization formula represents the lifelines of the diagrams. After the encoding of d is completed, these 7 formulae will be combined in a single formula that is going to be monitored.

Definition 7 *A* positive constraint (resp. negative constraint) *for a UML diagram d is a 2D-LTL formula that must be satisfied (respectively constantly dissatisfied) by the execution of a system which meets the UML specification described in d. Constraints describing interaction diagram d are denoted by $[|d|]$ where:*

$$\begin{aligned}
[|d|] &= \left\langle [|d|]_b^+, [|d|]_e^+, [|d|]_i^+, [|d|]_b^-, [|d|]_e^-, [|d|]_i^-, [|d|]_l \right\rangle \\
[|d|]_b^{+/-} &= \textit{true/false immediately when the interaction begins} \\
[|d|]_e^{+/-} &= \textit{true/false immediately when the interaction ends} \\
[|d|]_i^{+/-} &= \textit{true/false/ during the whole process of the execution} \\
[|d|]_l &= \textit{enumerates all the lifelines present in the diagram}
\end{aligned}$$

Our goal is to find such set of constraints that the following conditions are satisfied:

- if for trace t history $\mathcal{H}(t)$ satisfies the constraints then a trace is *well-formed*, that is the transmitter always precedes the receiver of the same message
- the trace of the execution t is positive for diagram d iff for all its prefixes

$t[..i]$ $\mathcal{H}(t[..i])$ satisfies the following 2D-LTL formula:

$$[|d|]^+ = H_G\ \left(\neg Y_L\ \top \rightarrow [|d|]_b^+\right) \wedge [|d|]_i^+ \wedge H_G\ \left(\neg Y_L\ [|d|]_e^+\right) \wedge H_G\ H_L\ [|d|]_l$$

- similarly, the trace of the execution t is negative iff for all its prefixes $t[..i]$ $\mathcal{H}(t[..i])$ satisfies the following formula:

$$[|d|]^- = H_G\ \left(\neg Y_L\ \top \rightarrow [|d|]_b^-\right) \wedge [|d|]_i^- \wedge H_G\ \left(\neg Y_L\ [|d|]_e^-\right) \wedge H_G\ H_L\ [|d|]_l$$

We will denote these accumulated formulae as $[|d|]^+$ and $[|d|]^-$ respectively.

Let us start defining these constraints from a primitive single-message diagram. A primitive diagram (message m between lifelines l_1 and l_2) is encoded so:

- the first global event must be the first event of some lifeline,
- the last global event must be the last event of one of the lifelines,
- the invariant requires that the transmitter of the message must precede the receiver,
- all negative constraints must be set to false,
- the synchronization formula is a simple enumeration of the lifelines.

Formally this is captured by the following formulae:

$$\begin{aligned}
[|d|]_b^+ &= l_1 \wedge !m \vee l_2 \wedge ?m \\
[|d|]_e^+ &= l_1 \wedge !m \vee l_2 \wedge ?m \\
[|d|]_i^+ &= O_G\ (l_2 \wedge ?m) \rightarrow O_G\ O_L\ (l_1 \wedge !m)
\end{aligned}
\qquad
\begin{aligned}
[|d|]_b^- &= \bot, \\
[|d|]_e^- &= \bot, \\
[|d|]_i^- &= \bot, \\
[|d|]_l &= l_1 \vee l_2
\end{aligned}$$

So the accumulated formula that defines the positive traces is

$$\begin{aligned}
[|d|]^+ = {} & H_G\ (\neg Y_L\ \top \rightarrow (l_1 \wedge !m \vee l_2 \wedge ?m)) \wedge \\
& \wedge (O_G\ (l_2 \wedge ?m) \rightarrow O_G\ O_L\ (l_1 \wedge !m)) \wedge \\
& \wedge H_G\ (\neg Y_L\ (l_1 \wedge !m \vee l_2 \wedge ?m)) \wedge H_G\ H_L\ (l_1 \vee l_2)
\end{aligned}$$

Let us now describe, how to represent weak sequencing by 2D-LTL formulae. Construction of the positive constraint is fairly simple. The precondition must be equivalent to the precondition of the first argument with addition of the first events from those second argument's lifelines, that are not involved in the first interaction. The postcondition can be constructed in a similar manner: it includes the last events of all the lifelines of the second argument together with the last events of those lifelines of the second argument that are not present in the first diagram. During the interaction the invariants of both arguments must hold, and the following additional constraint is introduced: if the first event of some lifeline of the second diagram occurs, and this lifeline participates in the first interaction, then the last event of the first interaction from the same lifeline must have already occurred.

To represent the alternative choice we have to introduce an auxiliary pair of predicates p_1, p_2. Only one of these predicates can be true in the same time, and since the execution of the diagram started its value cannot be changed. These predicates serve to synchronize precondition, postcondition and invariant. The predicate that is true denotes, which branch of two was chosen in the current execution.

We illustrate the algorithm on an example from Figure 11.

Example 8 *Let us calculate positive constraints $[|d|]_b^+$, $[|d|]_i^+$, $[|d|]_e^+$ and a constraint $[|d|]_l$ for diagram d representing the behavior of the game applet when limit of MMS messages was exceeded:*

$$\begin{aligned}
[|d|]_b^+ &= user \wedge ?ask \vee game \wedge MMS\_limit\_exceeded \\
[|d|]_e^+ &= p_1 \wedge user \wedge !yes \vee \\
&\vee p_1 \wedge game \wedge resume \vee \\
&\vee p_2 \wedge user \wedge !no \vee \\
&\vee p_2 \wedge game \wedge terminate \\
[|d|]_i^+ &= (Y_L \ (game \wedge MMS\_limit\_exceeded) \rightarrow game \wedge suspend) \wedge \\
&\wedge (Y_L \ (game \wedge suspend) \rightarrow game \wedge !ask) \wedge \\
&\wedge (p_1 \wedge (Y_L \ (game \wedge ?yes) \rightarrow game \wedge resume) \vee \\
&\vee \ p_2 \wedge (Y_L \ (game \wedge ?no) \rightarrow game \wedge terminate)) \wedge \\
&\wedge (Y_L \ (user \wedge ?ask) \rightarrow (p_1 \wedge user \wedge !yes \vee p_2 \wedge user \wedge !no)) \\
&\wedge (Y_L \ (game \wedge !ask) \rightarrow (p_1 \wedge game \wedge ?yes \vee p_2 \wedge game \wedge ?no)) \wedge \\
&\wedge (O_G \ (user \wedge ?ask) \rightarrow O_G \ O_L \ (game \wedge !ask)) \wedge \\
&\wedge (O_G \ (game \wedge ?yes) \rightarrow O_G \ O_L \ (user \wedge !yes)) \wedge \\
&\wedge (O_G \ (game \wedge ?no) \rightarrow O_G \ O_L \ (user \wedge !no)) \\
[|d|]_i^+ &= game \vee user
\end{aligned}$$

To monitor if the trace of execution is positive for UML specification defined by diagram d we must use an accumulated formula

$$[|d|]^+ = H_G \ \left(\neg Y_L \ \top \rightarrow [|d|]_b^+\right) \wedge [|d|]_i^+ \wedge H_G \ \left(\neg Y_L \ \ [|d|]_e^+\right) \wedge H_G \ H_L \ \ [|d|]_l$$

7 Related work

Run-time monitors are security policy enforcement mechanisms that work by monitoring execution steps of a system, called the target, and performing some specified actions (e.g. terminating the target's execution) if it is about to violate the security policy. They can be loosely classified as follows:

Each instance of every application is monitored individually. This approach by Erlingsson and Schneider [5] is simple and enables full in-lining of the monitor in the code of the program. However, some useful security policies, such as "one-out-of-k" by Edjlali et al.[3], cannot be captured.

All instances of each application are monitored at once. This approach is used in Krukow et al.'s work [12] and makes history-based decisions possible. Yet, monitoring the interactions among applications remains impossible.

All instances of all applications are monitored globally. The last approach behind our proposal has the advantage of being suitable for handling application interactions. It is very important to control this kind of properties in the mobile device environment, where applets must exchange information between each other or with the system. However, this approach makes full monitor in-lining difficult.

Schneider [19] introduced the notion of a *security automaton*, which takes as input a program's requested actions and determines whether a legal transition can be made from the current state. If no transition can be made, then the requested action is illegal and the target program is terminated. Security policies are represented by automata: easy to inline and process, less easy to write. Additional practical details can be found in Erlingsson's PhD Thesis [4]. Ligatti et al.[14] extended the automaton's behavior. They represent it as a transformation mechanism that can *edit* the stream of actions produced by the target. Their *edit automaton* can either terminate the target in response on illegal actions or modify its execution in order to respect the desired property.

Run-time monitoring can be applied not only for comparing a program's behavior with a prescribed one. It is also widely used for implementing various history-based policies. For instance, Fong [6] uses monitors to track a *shallow history* of previously granted access events. Such monitors can enforce useful policies, such as Chinese Wall and one-out-of-k [3].

Besides a mechanism for enforcement we need a language for policy writing, which is expressive enough to handle real-life policies and formal enough to enable effective enforcement [20]. Yet, writing directly a security automaton for a given security property is not easy. When looking for alternatives it is worth noting that only *safety properties* [19] can be enforced by monitors because monitors observe only single executions and cannot speculate on future executions. For instance, access control restrictions define safety properties, but not information flow (it does not mean that it cannot be controlled by other means [2,18]).

Thus, pure-past Linear Temporal Logic (pLTL) seems to be a good candidate for these properties. The idea and the practical implementation of run-time monitoring based on the recursive evaluation of pLTL formulae belong to Havelund and Roşu [9]. They write policies as pLTL formulae with predicates depending on the state of the execution, propositional and temporal logic operators and proposed an efficient way to monitor a program by using the recursive semantics of pLTL. Yet, they consider only single executions of the program.

Krukow *et al.* [12] extend this idea to multiple executions by replacing the notion of event with the notion of session, which intuitively corresponds to a single run of the program. A session is a set of events composed according to certain rules but the order of events within a session is not recorded. The intuitive distinction between event and session is that a session may be possibly updated with events even after the succeeding session has started. So this model is close to a single threaded suspend-and-resume execution of "concurrent" processes. Temporal operators are used for policy writing, but are applied to sessions rather than to events.

In [12] Krukow et al. make an example of a eBay bidder policy bid only to the auctions where the seller has never provided negative feedback in auctions where payment was made. To capture this policy by their language they introduce the *ignore* event that states that the bidder did not make payment. This policy would be more naturally formulated as bid only to the auctions where the seller has never provided negative feedback *after* payment was made. Yet this policy cannot be captured by their language.

While Havelund and Roşu allow any computable predicates on the execution states in pLTL formulae, Krukow et al. restrict them to the statements concerning the presence of events in the session. These latter statements are computationally convenient but are not enough for expressing useful policies. For this reason Krukow et al. extend their language and monitoring algorithm to handle parameterized events, quantified and quantitative properties.

Among the attempts to simplify policy writing, Hoagland et al.in [11] propose to specify access control policies in a graph-based policy language. However, hey do not provide the possibility to reason about temporal behavior of the program. Further, this proposed language does not rely on an widely used standard.

8 Conclusions

Venkatakrishnan et al.[20] enumerates a number of desired features of a policy framework for mobile applications: flexibility to state policies in terms of externally observable operations, ability to express policies involving temporal sequencing of operations, modular specifications with precise and simple semantics, and efficient enforcement.

Our framework has these features because it is based on a refined bi-dimensional model. It allows distinguishing between local and global policy constraints and therefore more fine-grained decision making with an efficiently enforceable monitoring mechanism. Technically, one could obtain the same results by using plain LTL and a global security monitor, which keeps track of all actions in a heap. However this solution leads to cumbersome, unreadable policies with a blow up of the formula due to the need of explicitly mentioning all sessions. Further it poses a limit on the maximum number of sessions that can be captured.

We also sketched how UML sequence diagrams can be mapped to 2D-LTL formulae, providing a possibility to specify properties in a more friendly graphical manner.

In the future we plan to study the optimizations of logical representation of policies by introducing different cost notions for predicates and in-lining the monitoring algorithm within the code..

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *10th Annual Network and Distributed System Symposium (NDSS'03)*, 2003.

[2] P. Bieber, J. Cazin, P. Girard, J. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. In *Proc. of ESORICS-00*, pages 1–16, 2000.

[3] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proc. of CCS-98*, pages 38–40. ACM Press, 1998.

[4] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Cornell University, 2004.

[5] U. Erlingsson and F. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. of NSPW-99*, 1999.

[6] P. Fong. Access control by tracking shallow execution history. In *Proc. of IEEE SSP-04*, pages 43–55. IEEE Press, 2004.

[7] O. Haugen, K. Husa, R. Runde, and K. Støélen. STAIRS towards formal design with sequence diagrams. *J. of Sys. and Software Mod.*, 2005.

[8] O. Haugen, R. Runde, and K. Stølen. How to transform UML neg into a useful construct. Technical report, University of Oslo, 2005.

[9] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Int. J. of STTT*, 2004.

[10] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 2004.

[11] J. A. Hoagland, R. Pandey, and K. Levitt. Security policy specification using a graphical approach. Technical Report CS-98-3, University of California, Dept. of Computer Science, Davis, California, 1998.

[12] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputationsystems with applications to historybased access control. In *Proc. of CCS-05*, 2005.

[13] D. Kushner. Engineering EverQuest: online gaming demands heavyweight data centers. *IEEE Spectrum*, 42(7):34–39, July 2005.

[14] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *IJIS*, 2003.

[15] F. Massacci and K. Naliuka. Towards practical security monitors of UML policies for mobile applications. Submitted to FOSSACS'07.

[16] Object Management Group. *UML 2.0 Superstructure Specification*, document: ptc/04-10-02 edition, 2004.

[17] J. Park and R. Sandhu. The $UCON_{ABC}$ usage control model. *TISSEC*, 7(1), 2004.

[18] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE JSAC*, 21(1), 2003.

[19] F. Schneider. Enforceable security policies. *J. ACM*, 3(1):30–50, 2000.

[20] V. N. Venkatakrishnan, R. Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *NSPW '02: Proceedings of the 2002 workshop on New security paradigms*, pages 61–68, New York, NY, USA, 2002. ACM Press.

[21] A. Zobel, C. Simoni, D. Piazza, X. Nuez, and D. Rodrguez. Business case and security requirements. Public Deliverable of EU Research Project D5.1.1, S3MS- Security of Software and Services for Mobile Systems, Report available at www.s3ms.org, October 2006.

A Proof of correspondence between serialized executions and histories

Proposition 1 *If $\mathcal{H}(t) = \langle S, s_f, \mathcal{T}, \mathcal{F}, L, P, V\rangle$ is a valid history then $\mathcal{H}(t \circ \langle e\rangle) = \left\langle S', s'_f, \mathcal{T}', \mathcal{F}', L', P', V'\right\rangle$is also a history.*

Proof. If e=new(l) then let us show that for $H' = \mathcal{H}(t \circ \langle e\rangle)$ all properties of a history hold:

(1) **Session past determinism** If $\mathcal{T}'(s) = t \circ \langle s_{-1}, s\rangle$ then
  (a) $s = s_0^{(l)}$
    $\mathcal{T}'(s) = \left\langle s_0^{(l)}\right\rangle \Rightarrow \mathcal{T}'(s) \neq t \circ \langle s_{-1}, s\rangle$ - the situation is impossible
  (b) $s \neq s_0^{(l)}$
    $\mathcal{T}(s) = \mathcal{T}'(s) = t \circ \langle s_{-1}, s\rangle \Rightarrow$ (because H is a history) $\mathcal{T}'(s_{-1}) = \mathcal{T}(s_{-1}) = t \circ \langle s_{-1}\rangle$
  Consequently the property of session past determinism holds.
(2) **Final past determinism** If $\mathcal{F}'\left(s'_f\right) = f_1 \circ \langle s'\rangle \circ f_2$ then
  (a) $s' = s_0^{(l)} = s'_f$
    $\mathcal{F}'\left(s'_f\right) = f_1 \circ \left\langle s'_f\right\rangle \circ \langle\rangle = f_1 \circ \left\langle s'_f\right\rangle$
  (b) $s \neq s_0^{(l)}$
    $\mathcal{F}'\left(s'_f\right) = \mathcal{F}(s_f) \circ \left\langle s'_f\right\rangle \Rightarrow \mathcal{F}(s_f) = f_1 \circ \langle s'\rangle \circ f'_2 \Rightarrow$
    $\mathcal{F}'(s') = \mathcal{F}(s') = f_1 \circ \langle s'\rangle$
(3) **Mutual past determinism** If $\mathcal{T}'(s) = t \circ \langle s_{-1}, s\rangle$, $\mathcal{F}'(s) = f \circ \langle s', s\rangle$ then
  (a) $s = s_0^{(l)}$
    $\mathcal{T}'(s) = \left\langle s_0^{(l)}\right\rangle \Rightarrow \mathcal{T}'(s) \neq t \circ \langle s_{-1}, s\rangle$ - the situation is impossible
  (b) $s \neq s_0^{(l)}$
    $\mathcal{T}(s) = \mathcal{T}'(s) = t \circ\langle s_{-1}, s\rangle, \mathcal{F}(s) = \mathcal{F}'(s) = f \circ\langle s', s\rangle \Rightarrow$ (because H is a history) exists $s'' \in \mathcal{T}(s') = \mathcal{T}'(s')$ such that $\mathcal{F}'(s_{-1}) = \mathcal{F}(s_{-1}) = f' \circ \langle s'', s_{-1}\rangle$
  Therefore the property of session past determinism holds.

Then let us check the properties if e=update(P_{new}, l_k):

(1) **Session past determinism** If $\mathcal{T}'(s) = t \circ \langle s_{-1}, s\rangle$ then
  (a) $s = s_{i+1}^{(l_k)}$
    $\mathcal{T}'\left(s_{i+1}^{(l_k)}\right) = \mathcal{T}\left(s_i^{(l_k)}\right) \circ \left\langle s_{i+1}^{(l_k)}\right\rangle = t \circ \left\langle s_i^{(l_k)}, s_{i+1}^{(l_k)}\right\rangle \Rightarrow$
    $\mathcal{T}'(s_{-1}) = \mathcal{T}'\left(s_i^{(l_k)}\right) = \mathcal{T}\left(s_i^{(l_k)}\right) = t$
  (b) $s \neq s_{i+1}^{(l_k)}$
    $\mathcal{T}(s) = \mathcal{T}'(s) = t \circ \langle s_{-1}, s\rangle \Rightarrow$ (because H is a history) $\mathcal{T}'(s_{-1}) = \mathcal{T}(s_{-1}) = t \circ \langle s_{-1}\rangle$

Consequently the property of session past determinism holds.

(2) **Final past determinism** If $\mathcal{F}'\left(s'_f\right) = \mathcal{F}\left(s_{i_{k-1}}^{(l_{k-1})}\right) \circ \left\langle s_{i+1}^{(l_k)}, \ldots, s_{i_n}^{(l_n)}\right\rangle = f_1 \circ \langle s' \rangle \circ f_2$ then

(a) $s' = s_{i_j}^{(l_j)}$, $j \geq k$
$f_1 = \mathcal{F}\left(s_{i_{k-1}}^{(l_{k-1})}\right) \circ \left\langle s_{i+1}^{(l_k)}, \ldots, s_{i_{j-1}}^{(l_{j-1})}\right\rangle$
$\mathcal{F}'(s') = \mathcal{F}\left(s_{i_{k-1}}^{(l_{k-1})}\right) \circ \left\langle s_{i+1}^{(l_k)}, \ldots, s_{i_j}^{(l_j)}\right\rangle \Rightarrow \mathcal{F}'(s') = f_1 \circ \left\langle s_{i_j}^{(l_j)}\right\rangle$

(b) $s' = s_{i_j}^{(l_j)}$, $j < k$
$\mathcal{F}(s_f) = \mathcal{F}\left(s_{i_{k-1}}^{(l_{k-1})}\right) \circ \left\langle s_i^{(l_k)}, \ldots, s_{i_n}^{(l_n)}\right\rangle = f_1 \circ \left\langle s_{i_j}^{(l_j)}\right\rangle \circ f''$, where $f'' = \left\langle s_{i_{j+1}}^{(l_{j+1})}, \ldots, s_i^{(l_k)}, \ldots, s_{i_n}^{(l_n)}\right\rangle$
$\Rightarrow$ (because H is a history) $\mathcal{F}'(s') = \mathcal{F}\left(s_{i_j}^{(l_j)}\right) = f_1 \circ \left\langle s_{i_j}^{(l_j)}\right\rangle$

Therefore the property of final past determinism holds.

(3) **Mutual past determinism** If $\mathcal{T}'(s) = t \circ \langle s_{-1}, s\rangle$, $\mathcal{F}'(s) = f \circ \langle s', s\rangle$ then

(a) $s = s_{i+1}^{(l_k)} \Rightarrow s_{-1} = s_i^{(l_k)}$, $s' = s_{i_{k-1}}^{(l_{k-1})}$
$s_i^{(l_k)} \in \mathcal{F}(s_f)$. By final past determinism property if $\mathcal{F}(s_f) = f_1 \circ \left\langle s_i^{(l_k)}\right\rangle \circ f_2$ then $\mathcal{F}\left(s_i^{l_k}\right) = f_1 \circ \left\langle s_i^{(l_k)}\right\rangle = f'_1 \circ \left\langle s_{i_{k-1}}^{(l_{k-1})}, s_i^{l_k}\right\rangle$ $\Rightarrow$ exists $s'' = s_{i_{k-1}}^{(l_{k-1})} \in \mathcal{T}'\left(s_{i_{k-1}}^{(l_{k-1})}\right)$ such that $\mathcal{F}'(s_{-1}) = f' \circ \langle s'', s_{-1}\rangle$

(b) $s = s_{i_{k+1}}^{(l_{k+1})} \Rightarrow s_{-1} = s_{i_{k+1}-1}^{(l_{k+1})}$, $s' = s_{i+1}^{(l_k)}$
$\mathcal{F}(s) = f \circ \left\langle s_i^{(l_k)}, s\right\rangle$, $\mathcal{T}(s) = \mathcal{T}'(s) = t \circ \langle s_{-1}, s\rangle \Rightarrow$ exists $s'' \in \mathcal{T}\left(s_i^{(l_k)}\right)$ such that $\mathcal{F}(s_{-1}) = \mathcal{F}\left(s_{i_{k+1}-1}^{(l_{k+1})}\right) = f' \circ \left\langle s'', s_{i_{k+1}-1}^{(l_{k+1})}\right\rangle$. But $\mathcal{T}'\left(s_{i+1}^{(l_k)}\right) = \mathcal{T}\left(s_i^{(l_k)}\right) \circ \left\langle s_{i+1}^{(l_k)}\right\rangle \Rightarrow s'' \in \mathcal{T}'\left(s_{i+1}^{(l_k)}\right) \Rightarrow$ exists $s'' \in \mathcal{T}'\left(s_{i+1}^{(l_k)}\right)$ such that $\mathcal{F}'(s_{-1}) = \mathcal{F}(s_{-1}) = f' \circ \langle s'', s_{-1}\rangle$.

(c) $s = s_{i_j}^{(l_j)}$, $k+1 < j \leq n \Rightarrow s_{-1} = s_{i_j-1}^{(l_j)}$, $s' = s_{i_{j-1}}^{(l_{j-1})}$
$\mathcal{F}(s) = f' \circ \left\langle s_{i_{j-1}}^{(l_{j-1})}, s\right\rangle$ where $f' = \mathcal{F}\left(s_{i_{j-2}}^{(l_{j-2})}\right)$, $\mathcal{T}(s) = \mathcal{T}'(s) = t \circ \langle s_{-1}, s\rangle \Rightarrow$ exists $s'' \in \mathcal{T}\left(s_{i_{j-1}}^{(l_{j-1})}\right)$ such that $\mathcal{F}(s_{-1}) = \mathcal{F}\left(s_{i_j-1}^{(l_j)}\right) = f'' \circ \left\langle s'', s_{i_j-1}^{(l_j)}\right\rangle$. But $\mathcal{T}'\left(s_{i_{j-1}}^{(l_{j-1})}\right) = \mathcal{T}\left(s_{i_{j-1}}^{(l_{j-1})}\right) \Rightarrow$ exists $s'' \in \mathcal{T}'(s')$ such that $\mathcal{F}'(s_{-1}) = \mathcal{F}(s_{-1}) = f'' \circ \langle s'', s_{-1}\rangle$.

(d) $s = s_{i_j}^{(l_j)}$, $1 < j < k \Rightarrow s_{-1} = s_{i_j-1}^{(l_j)}$, $s' = s_{i_{j-1}}^{(l_{j-1})}$
$\mathcal{T}(s) = \mathcal{T}'(s) = t \circ \langle s_{-1}, s\rangle$, $\mathcal{F}(s) = \mathcal{F}'(s) = f \circ \langle s', s\rangle \Rightarrow$ (because H is a history) exists $s'' \in \mathcal{T}(s') = \mathcal{T}'(s')$ such that $\mathcal{F}'(s_{-1}) = \mathcal{F}(s_{-1}) = f' \circ \langle s'', s_{-1}\rangle$

Therefore the property of session past determinism holds.

If e=end(l) then the properties are obviously preserved because the history does not change. $\square$

B Correctness of the monitoring algorithm

As stated for the proof of the main theorem we must first prove a technical lemma concerning the correctness of the **Evaluate** function (Figure 7).

Lemma 1 *Let H and $\mathcal{F}(s_f) = \langle s_1, \ldots, s_l \rangle$ and $\mathcal{T}(s_j) = t[..i_j]$ for some $s_j \in \mathcal{F}(s_f)$. Then if*

- *pre_loc[0..m] array contains the evaluation of all subformulae of 2D-LTL formula in state $t[..i_j - 1], \mathcal{F}(t[..i_j - 1])$. Subformulae are placed in such order that if now[k_1] is a subformula of now[k_2] then $k_1 > k_2$. If $i_j = 1$ then pre_loc[0..m]= nil,*
- *pre_gl[0..m] contains the evaluation of the formula in state $\mathcal{T}(s_{j-1}), \mathcal{F}(s_{j-1})$. If $j = 1$ then pre_gl[0..m]= nil,*
- *P is a set of predicates true in s_j, or $P = nil$ and predicates values are stored in the corresponding elements of now[0..m].*

*Then after executing **Evaluate** (Figure 7) now[0..m] array contains the evaluation of the formula in state $\mathcal{T}(s_j), \mathcal{F}(s_j)$.*

Proof. We will prove the lemma by structural induction. First let us assume that the element of the array now[k] corresponds to the subformula $\phi = p$, $p \in P$ that is ϕ is a predicate. Let us evaluate this subformula in s_j. By recursive semantics (see Figure 5) $\mathcal{T}(s_j), \mathcal{F}(s_j) \models \phi$ iff p holds in s_j. If $P \neq nil$ then predicates that hold in this state are stored in P. Then by steps 17-18 of **Evaluate** value of predicate p in state s_j is stored in now[k]. From the other hand if $P = nil$ the value of the predicate p in now[k] is left unchanged (step 16). Thus the base of induction is proven.

If the element of the array now[k_1] corresponds to the subformula $\phi = \neg\psi$ then there is an element now[k_2], $k_2 > k_1$, which contains the value of the subformula ψ in s_j. As evaluation proceeds from greater indexes to smaller this element will be already evaluated by the time when we evaluate now[k_1]. But $\mathcal{T}(s_j), \mathcal{F}(s_j) \models \neg\psi$ iff $\mathcal{T}(s_j), \mathcal{F}(s_j) \not\models \psi$. Therefore by step 20 of **Evaluate** the value of subformula ϕ is stored in now[k_1]. Similarly we can prove the cases when now[k_1] stores value of subformulae $\phi = \phi_1 \wedge \phi_2$ or $\phi = \phi_1 \vee \phi_2$ by observing lines 22, 24 of the algorithm.

If the element of the array now[k_1] corresponds to the subformula $\phi = Y_L\ \psi$ then there is an index k_2, $k_2 > k_1$ such that elements of arrays with this index store the value of subformula ψ. If $i_j > 1$ we say that $\mathcal{T}(s_j), \mathcal{F}(s_j) = t[..i_j], f[..j] \models Y_L\ \psi$ iff $t[..i_j - 1], \mathcal{F}(t[i_j - 1]) \models \psi$. But the values of the subformulae evaluated in state $t[..i_j - 1], \mathcal{F}(t[i_j - 1])$ are stored in the pre_loc[0..m] array. Therefore at step 27 we put a correct evaluation of ϕ in now[k_1]. Else if $i_j = 1$ then $s_j \not\models Y_L\ \psi$. In this case pre_loc[0..m]= nil and by step 26 of the algorithm we store false in now[k_1].

Similarly we can prove the case when now[k_1] stores the subformula $\phi = Y_G\ \psi$. By recursive semantic $\mathcal{T}(s_j), \mathcal{F}(s_j) = t[..i_j], f[..j] \models Y_L\ \psi$ iff $\mathcal{T}(f[j-1])$, $f[..j-1] = \mathcal{T}(s_{j-1}), \mathcal{F}(s_{j-1}) \models \psi$. Then we must only notice that array pre_gl[0..m] stores the values of subformulae evaluated in $\mathcal{T}(s_{j-1}), \mathcal{F}(s_{j-1})$, and the rest of the proof is identical to the previous case (see steps 29-30 of **Evaluate**).

For $\phi = \phi_1\ S_L\ \phi_2$, $\phi = \phi_1\ S_G\ \phi_2$ we apply exactly the same argument (recall that the recursive semantic allows evaluating these subformulae by referring only to states $t[..i_j - 1], \mathcal{F}(t[i_j - 1])$ and $\mathcal{T}(s_{j-1}), \mathcal{F}(s_{j-1})$ respectively). The result of the evaluation is also stored in now[k_1] (see lines 32-32 and 35-36 of the **Evaluate** function).

As this computation is performed for all elements of now[0..m] array, in the end of the function it stores the correct result of evaluation in state s_j. $\square$

The next and key part of the proof is an invariant property of the monitoring algorithm from Figure 8).

Lemma 2 *Let $\langle e_1, \ldots, e_n\rangle$ be a serialized trace of execution and let also $H = \mathcal{H}(\langle e_1, \ldots, e_n\rangle)$ be a corresponding history on it according Def. 6. Let also a number of sessions in the execution be equal to l_f and $\mathcal{F}(s_f) = \langle s_1, \ldots, s_l\rangle$. Then if now[1..$l_f$][0..m], pre[1..$l_f$][0..m] are arrays created by algorithm* Monitor *for evaluation of a 2D-LTL function ϕ then for all $j = 1..l_f$ the following properties hold:*

(1) now[j] contains the result of the formula evaluation in state s_j,
(2) if $\mathcal{T}(s_j) = t[..i_j]$ then pre[j] contains the result of the formula evaluation in state $s_j^{(-1)} = t[i_j - 1]$,
(3) now[l_f][0..m] is evaluated in s_f.

Proof. To prove that this invariant is maintained by the algorithm we will use induction. The base of induction, for the empty trace, is obvious because there is no array and therefore the claim is vacuously true.

For the inductive case we assume that (1)-(3) hold for trace $\langle e_1, \ldots, e_n\rangle$ and show that they hold for $\langle e_1, \ldots, e_n, e_{n+1}\rangle$.

If $e_{n+1} = new(l_f + 1)$ then by construction of $H' = \mathcal{H}(\langle e_1, \ldots, e_n, e_{n+1}\rangle)$ we have a new final state $s'_f = s_{l_f+1}$ and $\mathcal{F}'\left(s_{l_f+1}\right) = \mathcal{F}\left(s_{l_f}\right) \circ \left\langle s_{l_f+1}\right\rangle$. Since $\mathcal{T}'(s) = \mathcal{T}(s)$, $\mathcal{F}'(s) = \mathcal{F}(s)$ for all $s \neq s_{l_f+1}$ claims (1) and (2) hold for $s_1, \ldots, s_{l_f}$. We must also show that they hold for $l_f + 1$.

By induction hypothesis the prerequisite of Lemma 1 hold. So we can conclude that after step 17 of the MONITORING algorithm (1) holds for s_{l_f+1}. Also pre$[l_f+1]$ is correctly equal to *nil* because there is no previous state so (2) also holds. Finally (3) holds because the final state of the new history is exactly s_{l_f+1}.

If $e_{n+1} = \mathit{update}(P, l)$ then by construction of the new history we add a new state s_l^*. For all s_j, $j = l+1, \ldots, l_f$ the frontier changes: in $\mathcal{F}(s_j)$ s_l is replaced with s_l^*. For this new state we have $\mathcal{T}'(s_l^*) = \mathcal{T}(s_l) \circ \langle s_l^* \rangle$, $\mathcal{F}'(s_l^*) = \mathcal{F}(s_l) \circ \langle s_l^* \rangle$. By construction for all s_j, $j < l$ we do not change any element of the frontiers: $\mathcal{F}'(s_j) = \mathcal{F}(s_j)$. So for these states (1) holds by the inductive hypothesis and therefore now$[l-1]$ contains the evaluation of the formula in state $\mathcal{T}'(s_{l-1}), \mathcal{F}'(s_{l-1})$.

By step 19 of the algorithm pre$[l]$ contains the evaluation of the formula in state $\mathcal{T}(s_l), \mathcal{F}(s_l)$ (because of the inductive hypothesis (1) applied to s_l). But by construction $\mathcal{T}'(s_l^*) = \mathcal{T}(s_l) \circ \langle s_l^* \rangle$ and therefore (2) holds for pre$[l]$ with respect to s_l^*.

Now the hypothesis of Lemma 1 holds for s_l^* and therefore after step 20 claim (1) also holds for s_l^*. Then by induction on j for $j = l+1 \ldots l_f$ the hypothesis of Lemma 1 holds for s_j. Then after execution of the cycle 21-22 claim (1) is true for all s_j. But (2) is also true since we have not changed neither $\mathcal{T}(s_j)$ nor pre$[j]$ and therefore the inductive step is performed for claims (1)-(2).

After step 22 the claim (3) also holds. Therefore the inductive step is completed. □

The proof of the main theorem follows from this lemma as simple corollary by observing that by the preceding Lemma $\mathit{now}[l_f][0]$ contains the truth value of the monitored formula in the final state s_f and the main algorithm throws an exception and terminates if the monitored formula is not satisfied.

C Correctness of the optimized monitoring algorithm

Lemma 3 *Let us assume that destroyed vectors continue to hold the values they stored at the time of destruction but simply cannot be accessed by the algorithm. Then we claim that the algorithm will only have to use*

- *pre[l], $l \in$*RunningSessions*,*
- *now[l], for all sessions in* $\mathbf{min}(\text{RunningSessions}) - 1 < l \leq l_f$

Proof. For (1) we simply note that by Definition 2 of a serialized trace no events of the form update(P,l) can occur after end(l) and therefore the algorithm does not make use of pre$[l]$ in any other place.

For (2) if the terminating session $l \neq$**min**(RunningSessions) then cycle at lines 29-30 will not run and therefore now$[l]$ will be still available. However if $l =$**min**(RunningSessions) then by Definition 2 we can only receive events of the form update(l') for $l' \geq$**min**(RunningSessions$-\{l\}$). So step 23 and cycle 24-25 will make use only of now$[j]$ for $l' - 1 \leq j \leq l_f$. Therefore step 30 will make inaccessible only those elements that the algorithm will not use.

The remaining argument of Lemma 2 will remain unchanged as the values now$[j]$, pre$[j]$ will be the correct ones needed by the inductive algorithm. $\square$