On Formalization of Operating Systems Behaviour Verification

Alexey Khoroshilov

Institute for System Programming of the Russian Academy of Sciences
Moscow, Russia
e-mail: khoroshilov@ispras.ru

ABSTRACT
Operating systems are responsible for correct implementation of computation environment properties that are usually assumed during verification of application software like virtual address space, scheduling, time management, etc. Formal analysis of implementation of these aspects is often implemented using specific models and verification techniques. As a result comprehensive verification of operating systems requires a systematic combination of various verification techniques applied for checking different properties. The paper proposes a generic approach to formalization of operating systems behavioural properties verification that allows to apply wide range of verification techniques and to support formal reasoning of their compositions.

Keywords
Operating systems, formal verification, behavioural specification.

1. INTRODUCTION

Behaviour of application software is usually formalized using concepts of programming languages like variables, heap, stack, data types, functions, etc. Models of programs written in low level languages like C may operate with low-level memory models such as an array of bytes. Models of concurrent program may take into consideration memory consistency issues, synchronous and asynchronous communications.

The task of formalization of operating systems behaviour is complicated by the fact that it is the operating system that is responsible for management of all the computation environment properties that are usually assumed to be correct during verification of application software. It includes management of virtual address spaces, process/thread migrations, scheduling, communication with devices via interrupts and direct memory access, etc.

Various approaches were proposed how to express low-level operational semantics of operating systems, e.g., using automaton hardware model of CPUs [1] or specialized imperative languages [2]. It enables verification of specific properties using specific methods up to leaving some aspects of reasoning in informal ground [3]. The problem was mitigated in the scope of particular projects [4,5], but there were no systematic solutions proposed.

The paper presents a more generic approach to formalize the operating system verification task that should allow to apply wider range of verification techniques and to support formal reasoning of their compositions.

Section 2 contains definitions of required terms and basic classification of behavioural properties. Section 3 describes how abstraction applies to verification task and presents an approach to verification of operating systems using different verification techniques. Section 4 wraps up conclusions and discusses possible future directions.

2. LOW LEVEL MODEL

2.1. Behavioural Model

Let us consider the behaviour of an operating system on a particular hardware in fixed configuration.

HWTrace is a set of potential events in hardware like a change of value in a register or in a memory cell, an interrupt event, a message sending b a bus, etc. An element of the set does not only represent a type of an event, but it also includes all attributes of the event such as values, timing, etc. So we assume that any two possible events are represented by different elements of **HWTrace**. There is an anti-reflexive binary relation $\prec_{dep}$ that means an explicit or implicit dependency between elements such as the second event can happen only later than the first one.

Event trace is a finitary partially-ordered set $(\mathbf{T}, \preccurlyeq)$, where

- $\mathbf{T} \subseteq$ **HWEvents** ;
- partial order $\preccurlyeq$ is a transitive and reflexive closure of $\prec_{dep}$, i. e. $\preccurlyeq = (\{(e,e)|e \in \mathbf{T}\} \cup \prec_{dep|\mathbf{T}})^*$, that defines the order of events in time and their dependencies between them;
- $\prec_{dep|\mathbf{T}} = \prec_{dep} \cap \mathbf{T}\times\mathbf{T}$ is a projection of $\prec_{dep}$ on $\mathbf{T}$;
- finitary means $\forall\, e \in \mathbf{T}$ the set $\{e' \in \mathbf{T} | e' \preccurlyeq e\}$ is finite.

Event trace represents a hardware execution, where $\mathbf{T}$ contains all events happened. Finitarity expresses the assumption that only finite number of events can happen during finite time. In different event traces the same events can be ordered or unordered, so there is no partial order on **HWEvents**. Instead there is a dependency relation $\prec_{dep}$ that can be non-transitive and even its transitive closure can be cyclic.

We will use the following denotations:

- $\wp(\mathbf{X})$ — set of all subsets of $\mathbf{X}$;
- $\wp_{\mathbf{po}}(\mathbf{X})$ — set of all partially-ordered subsets of $\mathbf{X}$;
- $\wp_{\mathbf{fpo}}(\mathbf{X})$ — set of all partially-ordered finitary subsets of $\mathbf{X}$;
- $\mathbf{Traces}(\mathbf{HWEvents}, \prec_{dep}) \subseteq \wp_{\mathbf{fpo}}(\mathbf{HWEvents})$ — set of all traces of **HWEvents** with a dependency relation $\prec_{dep}$;
- $\wp_{\mathbf{directed}}(\mathbf{X}, \preccurlyeq) = \{\mathbf{X'} \subseteq \mathbf{X} | \ \forall\, \mathbf{T_1},\mathbf{T_2} \in \mathbf{X'} \ \exists\, \mathbf{T_3} \in \mathbf{X'}: \mathbf{T_1} \preccurlyeq \mathbf{T_3} \wedge \mathbf{T_2} \preccurlyeq \mathbf{T_3})\}$ — set of all directed subsets of $(\mathbf{X}, \preccurlyeq)$;
- $\wp_{\mathbf{dcpo}}(\mathbf{X}, \preccurlyeq) = \{\mathbf{X'} \in \wp(\mathbf{X}) | \ \forall\, \mathbf{Y} \in \wp_{\mathbf{directed}}(\mathbf{X'}, \preccurlyeq|_{\mathbf{X'}}) \ \mathbf{sup}(\mathbf{Y}) \in \mathbf{X'}\}$ — set of all directed complete partially ordered subsets of $\mathbf{X}$;
- $\mathbf{Prefix}(\mathbf{T}, \preccurlyeq) = \{(\mathbf{T'}, \preccurlyeq') \mid \mathbf{T'} \subseteq \mathbf{T} \wedge \forall\, e \in \mathbf{T} \ \forall\, e' \in \mathbf{T'} \ (e \preccurlyeq e') \Rightarrow e \in \mathbf{T'} \wedge \preccurlyeq' = \preccurlyeq|_{\mathbf{T'}}\}$ - set of all prefixes of partially ordered set $(\mathbf{T}, \preccurlyeq)$;
- $\preccurlyeq_{pre} = \{(\mathbf{T_1},\mathbf{T_2}) \mid \mathbf{T_1} \in \mathbf{Prefix}(\mathbf{T_2})\}$ — is_prefix relation;
- $\wp_{\mathbf{cleft}}(\mathbf{X}) = \{\mathbf{X'} \in \wp_{\mathbf{dcpo}}(\mathbf{X}, \preccurlyeq_{pre}) \mid \forall\, (\mathbf{T}, \preccurlyeq) \in \mathbf{X'} \ \mathbf{Prefix}(\mathbf{T}, \preccurlyeq) \subseteq \mathbf{X'}\}$ — set of all **dcpo**-subsets of $(\mathbf{X}, \preccurlyeq_{pre})$ left-closed w.r.t. partial orders of its elements.

A set of all possible event traces for $\mathbf{(HWEvents, \prec_{dep})}$ is $\mathbf{HWTraces} \in \wp_{\mathbf{cleft}}(\mathbf{Traces(HWEvents, \prec_{dep})})$, where all possible means that all the traces are compliant to hardware specification. Thus, **HWTraces** represents a model of the hardware managed by an operating system.

If there is an infinite sequence of event traces in **HWTraces** such that each next trace extends the previous one then supremum of the sequence also is in the **HWTraces**. That is why it is required **HWTraces** to be a **dcpo-**subset. If an event trace is possible that any its prefix is possible, so it is required **HWTraces** to be left-closed w.r.t. partial orders of its elements.

There is a number of formalisms suitable for compact description of event trace sets including Petri nets [6], flow event structures [7], event structures [8], Pomsets [9], process algebras like CCS, CSP, ACP [10,11,12].

Behavioural model of an operating system that manages hardware can be considered as a subset of all possible event traces of the given hardware platform $\mathbf{HWTraces_{OSImage}} \in \wp_{\mathbf{cleft}}(\mathbf{Traces(HWEvents, \prec_{dep})})$, where $\mathbf{HWTraces_{OSImage}} \subseteq \mathbf{HWTraces}$.

Behavioural property of operating system is defined in a straightforward way as a set of sets of event traces $\mathbf{OSValid} \subseteq \wp_{\mathbf{cleft}}(\mathbf{HWTraces})$. Behavioural model of an operating system is considered to have a property **OSValid** iff $\mathbf{HWTraces_{OSImage}} \in \mathbf{OSValid}$.

So far, we defined the behavioural model of operating systems in terms of collective trace semantics of events happened in hardware level. As far as the given trace semantics preserves concurrency of events, it can be used with verification techniques based on both true concurrency models and interleaved models.

2.2. Abstract Models

Following the ideas of abstract interpretation [13] let us define a concept of abstraction.

Let us denote a lattice of all subsets of $\wp_{\mathbf{cleft}}(\mathbf{HWTraces})$ as $\mathbb{L} \equiv (\wp(\wp_{\mathbf{cleft}}(\mathbf{HWTraces})), \subseteq, \cup, \cap)$. Then an *abstraction* $(\mathbb{A}, \gamma)$ is a pair $(\mathbb{A}, \gamma)$, where $\mathbb{A}$ – an abstract domain, and $\gamma: \mathbb{A} \to \mathbb{L}$ is an injective function.

Partial order $\subseteq$ of $\mathbb{L}$ defines the partial order $\sqsubseteq$ of $\mathbb{A}$ $a_1 \sqsubseteq a_2 \equiv \gamma(a_1) \subseteq \gamma(a_2)$ since γ is injective. Function of *best over-approximation* $\alpha: \mathbb{L} \to \mathbb{A}$ maps the precise property $r \in \mathbb{L}$ to an abstract property $\alpha(r) \in \mathbb{A}$:

- $r \subseteq \gamma(\alpha(r))$ over-approximation
- $\forall\, a \in \mathbb{A}\ r \subseteq \gamma(a) \Rightarrow \gamma(\alpha(r)) \subseteq \gamma(a)$ best over-approximation

It is known [10] that there exists a function of *best over-approximation* α iff posets $\mathbb{L}$ and $\mathbb{A}$ with functions α and γ compose a Galois connection $(\alpha,\gamma): \mathbb{L} \leftrightharpoons \mathbb{A}$.

2.3. Properties Classification

Property $\mathbf{R_m} \in \mathbb{L}$ is called a *negative property* iff $\forall\, P \in \wp_{\mathbf{cleft}}(\mathbf{HWTraces})\ P \notin \mathbf{R_m} \Rightarrow \forall\, P' \in \wp_{\mathbf{cleft}}(\mathbf{HWTraces})\ P \subseteq P' \Rightarrow P' \notin \mathbf{R_m}$. Class of negative properties **HNeg** is the most practically useful one among all properties in collective trace semantics.

Dual class of *positive properties* **HPos** includes properties $\mathbf{R_m} \in \mathbb{L}$: $\forall\, P \in \mathbf{R_m}\ \ \forall\, P' \in \wp_{\mathbf{cleft}}(\mathbf{HWTraces})\ \ P \subseteq P' \Rightarrow P' \in \mathbf{R_m}$.

Let us define an auxiliary function **Pchar** that maps a set of event traces P and a property $\mathbf{R_m} \in \mathbb{L}$ to $\mathbf{Pchar}(P, \mathbf{R_m}) = \{TS \subseteq P |\ \mathbf{Prefixes}(TS) \notin \mathbf{R_m} \wedge \forall\, TS' \subset TS\ \mathbf{Prefixes}(TS') \in \mathbf{R_m}\}$, where $\mathbf{Prefixes}(TS) = \{t \in \mathbf{HWTraces} |\ \exists\ t' \in TS: t \in \mathbf{Prefix}(t')\}$ — a union of all prefixes of traces from $TS \in \wp(\mathbf{HWTraces})$. Let us denote $\mathbf{Pchar_k}(P, \mathbf{R_m}) = \{TS \in \mathbf{Pchar}(P, \mathbf{R_m}) |\ |TS| = k\}$. Then $\mathbf{R_m} \in \mathbf{HNeg}$ is called a *negative property of cardinality k* $\mathbf{HN_k}$ iff $\forall\, P \in \wp_{\mathbf{cleft}}(\mathbf{HWTraces}) \setminus \mathbf{R_m}\ \ \mathbf{Pchar_k}(P, \mathbf{R_m}) \neq \varnothing$.

Well-known abstraction of collective trace semantics is the individual trace semantics. Class of properties precisely represented in individual trace semantics $\mathbf{H_{join}}$ is a subclass of negative properties, where
$\mathbf{H_{join}} = \{\mathbf{R_m} \in \wp(\wp_{\mathbf{cleft}}(\mathbf{HWTraces})) |\ \exists\, \mathbf{P_m} \subseteq \mathbf{HWTraces}: \forall\, P \in \wp_{\mathbf{cleft}}(\mathbf{HWTraces})\ P \in \mathbf{R_m} \Leftrightarrow P \subseteq \mathbf{P_m}\}$.
Moreover, $\mathbf{H_{join}}$ equals the class of negative properties of cardinality 1 $\mathbf{HN_1}$.

3. OPERATING SYSTEM VERIFICATION

3.1. Verification in Abstraction

The task of verification is to check if a behavioural model of an operating system $\mathbf{HWTraces_{OSImage}}$ has a property **R**, i.e., $\mathbf{HWTraces_{OSImage}} \in \mathbf{R}$. As far as low level model is too detailed and too complex, abstraction is a typical approach to overcome those problems.

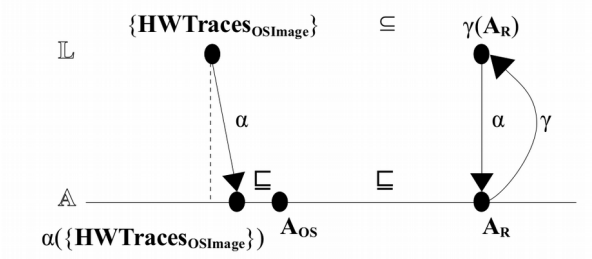
For an abstraction $(\alpha,\gamma): \mathbb{L} \leftrightharpoons \mathbb{A}$ the approach looks as follows. Behavioural model $\mathbf{HWTraces_{OSImage}}$ is represented by $\mathbf{A_{OS}} \in \mathbb{A}$ such that $\alpha(\{\mathbf{HWTraces_{OSImage}}\}) \sqsubseteq \mathbf{A_{OS}}$. If the property **R** has a precise representation in $\mathbb{A}$ $\mathbf{A_R} \in \mathbb{A}$ then the verification task can be reduced to checking $\mathbf{A_{OS}} \sqsubseteq \mathbf{A_R}$.

There are two kinds of potential problems of verification methods: unsoundness and imprecision.

- unsoundness means situations when verification misses actual bugs;
- imprecision means situations when verification reports false alarms.

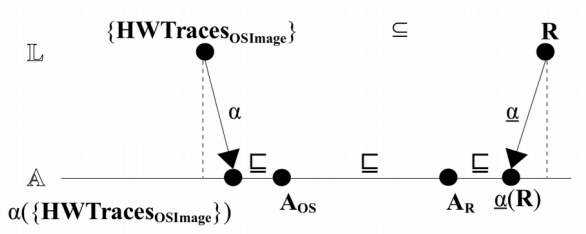
The assumptions made guarantee that the approach above is sound: $\mathbf{A_{OS}} \sqsubseteq \mathbf{A_R} \Rightarrow \alpha(\{\mathbf{HWTraces_{OSImage}}\}) \sqsubseteq \mathbf{A_{OS}} \sqsubseteq \mathbf{A_R} \Rightarrow \gamma(\alpha(\{\mathbf{HWTraces_{OSImage}}\})) \subseteq \gamma(\mathbf{A_{OS}}) \subseteq \gamma(\mathbf{A_R}) \Rightarrow \{\mathbf{HWTraces_{OSImage}}\} \subseteq \gamma(\alpha(\{\mathbf{HWTraces_{OSImage}}\})) \subseteq \gamma(\mathbf{A_R}) \Rightarrow \mathbf{HWTraces_{OSImage}} \in \gamma(\mathbf{A_R})$.

Lack of precision happens if $\mathbf{A_{OS}} \neq \alpha(\{\mathbf{HWTraces_{OSImage}}\})$ (Picture 1). Gap between $\alpha(\{\mathbf{HWTraces_{OSImage}}\})$ and $\mathbf{A_{OS}}$ may lead to false alarms.



Picture 1. Verification of precisely representable property

If the property **R** does not have a precise representation in $\mathbb{A}$ ($\gamma(\alpha(\mathbf{R})) \neq \mathbf{R}$) then the picture becomes more complex (Picture 2).



Picture 2. Verification of precisely non-representable property

To keep soundness it is required to choose $\mathbf{A}_{\mathbf{R}} \in \mathbb{A}$ such that $\mathbf{A}_{\mathbf{R}} \sqsubseteq \underline{\alpha}(\mathbf{OSReq})$ and to check if $\mathbf{A}_{\mathbf{OS}} \sqsubseteq \mathbf{A}_{\mathbf{R}}$. Here $\underline{\alpha}$ is a function of best under-approximation dual to α. The problem of imprecision in this case can be very significant up to missing any sense. For example, properties not belonging to **HNeg** are mapped by $\underline{\alpha}$ for individual trace semantics abstraction to $\varnothing$ and check if $\mathbf{A}_{\mathbf{OS}} \sqsubseteq \varnothing$ is pointless. Thus, it is highly desirable to choose abstraction that allows to represent target property precisely.

3.2. Abstraction and Property Classes

An important requirement for soundness of verification is to choose the approximation $\mathbf{A}_{\mathbf{OS}}$: $\alpha(\{\mathbf{HWTraces}_{\mathbf{OSImage}}\}) \sqsubseteq \mathbf{A}_{\mathbf{OS}}$ or $\gamma(\alpha(\{\mathbf{HWTraces}_{\mathbf{OSImage}}\})) \subseteq \gamma(\mathbf{A}_{\mathbf{OS}})$. This means $\mathbf{A}_{\mathbf{OS}}$ has to represent the exact set of event traces that operating system can produce optionally with other set of traces. This requirement can be relaxed for some kinds of properties.

First of all, let us consider a class of properties $\mathbf{HClass} \subseteq \wp(\wp_{\mathbf{cleft}}(\mathbf{HWTraces}))$. It induces abstraction $(\mathbb{A}, \gamma_{\mathbf{HClass}}) = (\mathbf{HClass}, id)$, where the abstract domain equals **HClass** and $\gamma_{\mathbf{HClass}}$ is an identity function. In this case $\sqsubseteq$ on $\mathbb{A}$ equals subset relation $\subseteq$:
$a_1 \sqsubseteq a_2 \equiv \gamma(a_1) \subseteq \gamma(a_2) = a_1 \subseteq a_2$.

If **HClass** is a complete lattice and $\mathbf{R}_{\mathbf{true}} = \wp_{\mathbf{cleft}}(\mathbf{HWTraces}) \in \mathbf{HClass}$, then there exists a function of the best over-approximation $\alpha_{\mathbf{HClass}}$: $\wp(\wp_{\mathbf{cleft}}(\mathbf{HWTraces})) \to \mathbf{HClass}$ that maps each property to a property from **HClass** representing the original one best of all. By the way, $\alpha_{\mathbf{HClass}}$ defines a preorder relation $\sqsubseteq_{\mathbf{HClass}}$: $R_1 \sqsubseteq_{\mathbf{HClass}} R_2 \equiv \alpha_{\mathbf{HClass}}(R_1) \subseteq \alpha_{\mathbf{HClass}}(R_2)$ that can be propagated to any other abstraction $(\mathbb{A}, \gamma)$: $a_1 \sqsubseteq_{\mathbf{HClass}} a_2 \equiv \gamma(a_1) \sqsubseteq_{\mathbf{HClass}} \gamma(a_2)$.

Theorem. If abstraction $(\mathbf{HClass}, id)$ has a function of the best over-approximation $\alpha_{\mathbf{HClass}}$, and the property **R** belongs to **HClass** ($\mathbf{R} \in \mathbf{HClass}$) then from existence of $\mathbf{A}_{\mathbf{OS}}$ such that $\alpha(\{\mathbf{HWTraces}_{\mathbf{OSImage}}\}) \sqsubseteq_{\mathbf{HClass}} \mathbf{A}_{\mathbf{OS}}$ and $\mathbf{A}_{\mathbf{OS}} \sqsubseteq \underline{\alpha}(\mathbf{R})$ follows correctness of the operating system regarding property **R** ($\mathbf{HWTraces}_{\mathbf{OSImage}} \in \mathbf{R}$).

Proof.
(1) $\mathbf{A}_{\mathbf{OS}} \sqsubseteq \underline{\alpha}(\mathbf{R}) \Rightarrow \gamma(\mathbf{A}_{\mathbf{OS}}) \subseteq \gamma(\underline{\alpha}(\mathbf{R})) \subseteq \mathbf{R} \Rightarrow \gamma(\mathbf{A}_{\mathbf{OS}}) \subseteq \mathbf{OSR}$.
(2) $\alpha(\{\mathbf{HWTraces}_{\mathbf{OSImage}}\}) \sqsubseteq_{\mathbf{HClass}} \mathbf{A}_{\mathbf{OS}}$
$\Rightarrow \{\mathbf{HWTraces}_{\mathbf{OSImage}}\} \sqsubseteq_{\mathbf{HClass}} \gamma(\mathbf{A}_{\mathbf{OS}})$
$\Rightarrow \alpha_{\mathbf{HClass}}(\{\mathbf{HWTraces}_{\mathbf{OSImage}}\}) \subseteq \alpha_{\mathbf{HClass}}(\gamma(\mathbf{A}_{\mathbf{OS}}))$
$\Rightarrow$ [$\alpha_{\mathbf{HClass}}$ — over-approximation]
$\{\mathbf{HWTraces}_{\mathbf{OSImage}}\} \subseteq \alpha_{\mathbf{HClass}}(\{\mathbf{HWTraces}_{\mathbf{OSImage}}\}) \subseteq$
$\subseteq \alpha_{\mathbf{HClass}}(\gamma(\mathbf{A}_{\mathbf{OS}}))$
$\Rightarrow$ [(1) and $\alpha_{\mathbf{HClass}}$ — over-approximation for $\gamma_{\mathbf{HClass}} = id$]
$\{\mathbf{HWTraces}_{\mathbf{OSImage}}\} \subseteq \alpha_{\mathbf{HClass}}(\gamma(\mathbf{A}_{\mathbf{OS}})) \subseteq \mathbf{R}$
$\Rightarrow \mathbf{HWTraces}_{\mathbf{OSImage}} \in \mathbf{R}$. □

For example, for **HClass** = **HNeg** the abstraction $(\mathbf{HNeg}, id)$ has a function of the best over-approximation $\alpha_{\mathbf{HNeg}}$:
$\alpha_{\mathbf{HNeg}}(\mathbf{R}) = \{P \in \wp_{\mathbf{cleft}}(\mathbf{HWTraces}) | \exists\, P' \in \mathbf{R}: P \subseteq P'\} = \downarrow\mathbf{R}$

Correspondingly, preorder $\sqsubseteq_{\mathbf{HNeg}}$ is as follows:
$R_1 \sqsubseteq_{\mathbf{HNeg}} R_2 \equiv \downarrow R_1 \subseteq \downarrow R_2 = \forall\, P \in R_1\ \exists\, P' \in R_2: P \subseteq P'$

For singletons it means $\{P_1\} \sqsubseteq_{\mathbf{HNeg}} \{P_2\} \equiv P_1 \subseteq P_2$.

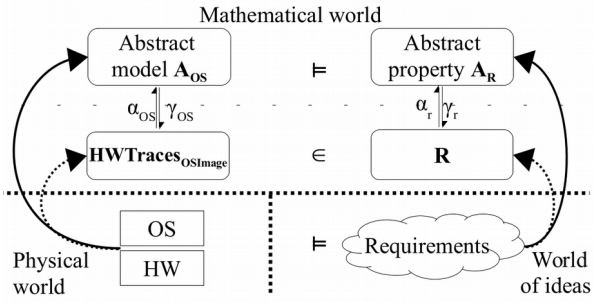
It justifies the conclusion obvious from intuition that if $\gamma(\mathbf{A}_{\mathbf{OS}})$ is a superset of a set of all event traces produced by target operating system then the approximation $\mathbf{A}_{\mathbf{OS}}$ could be soundly used for checking properties of negative class **HNeg**.

3.3. Operating Systems Verification

Let us consider a task of operating system verification against a set of requirements $\mathbf{OSValid} \subseteq \wp_{\mathbf{cleft}}(\mathbf{HWTraces})$. It is suggested to decompose the requirements as a finite hierarchy of properties, where leaf properties are convenient for some verification method.

A parent-child relation in the hierarchy means that the child property is a subproperty of the parent property ($\mathbf{R}_{\mathbf{parent}} \subseteq \mathbf{R}_{\mathbf{child}}$), and the intersection of all the children should cover the parent property ($\mathbf{R}_{\mathbf{parent}} = \cap\mathbf{R}_{\mathbf{child}}$). As a result checking of all leaf properties will lead to the checking of the root property.

Picture 3 demonstrates the interaction between the main entities participating in verification of a leaf property. Operating system and hardware exist in a physical world and they can be "felt" in some sense. The property of the operating system and highly coupled concept of "satisfy" relation $\models$ cannot be easily "felt", while many properties have quite transparent representation in the physical world.



Picture 3. Overview of operating system verification

Low-level models $\mathbf{HWTraces}_{\mathbf{OSImage}}$, **R** and $\in$ represent in mathematical world a composition of operating system and hardware, requirements to operating system and "satisfy" relation, correspondingly. But these mathematical objects exist hypothetically, since it is too complex to construct them. Actual analysis is applied to abstract behavioural model $\mathbf{A}_{\mathbf{OS}}$, abstract property $\mathbf{A}_{\mathbf{R}}$ and relation "satisfy" between them. These mathematical objects ignores many details of low-level models.

While $\mathbf{A}_{\mathbf{OS}}$ and $\mathbf{A}_{\mathbf{R}}$ can be elements of different abstractions, we consider the case when $\mathbf{A}_{\mathbf{OS}}$ and $\mathbf{A}_{\mathbf{R}}$ are elements of the same abstract domain $\mathbb{A}$ with partial order $\sqsubseteq$ playing the role of the "satisfy" relation $\models$. There is a hypothetical mapping between the abstract domain $\mathbb{A}$ and the low-level models $\gamma : \mathbb{A} \to \mathbb{L}$ and, possibly, a function of the best over-approximation α: $\mathbb{L} \to \mathbb{A}$. Ideally, the abstract domain should

be chosen so that it is able to precisely represent the target property **R** as $\mathbf{A_R} \in \mathbb{A}$ and there is a function of the best over-approximation α.

The target property is formalized as $\mathbf{A_R}$. There is a need to check if $\mathbf{A_R}$ correctly represents **R**, but in general case it can only be done mentally.

The next step is to choose an element $\mathbf{A_{OS}} \in \mathbb{A}$ that soundly represents the target operating system. To simplify the analysis of this fact, the choice can be done taking into account specifics of the target property. It may allow to relax the requirements to the relation between the low-level behavioural model and the abstract one:
$\alpha(\{\mathbf{HWTraces_{OSImage}}\}) \sqsubseteq_{\mathbf{HClass}} \mathbf{A_{OS}}$ if $\mathbf{R} \in \mathbf{HClass}$.

Finally, the original verification task of checking $\mathbf{HWTraces_{OSImage}} \in \mathbf{R}$ is reduced to the check $\mathbf{A_{OS}} \sqsubseteq \mathbf{A_R}$ that can be formally done using some verification method.

Any of the steps above can produce additional assumptions or limitations of verification methods that should be analyzed and verified separately. It can be considered that these extra properties are added to the target hierarchy of requirements and are verified using other verification methods according to the same scheme.

Using a property from the hierarchy for proving another property should be done to avoid cyclic dependencies between them. Such cyclic dependencies can happen sometimes, for example, if the dependencies are only used for induction step, but validness of the dependencies should be carefully managed.

The proposed approach for operating system verification using different verification techniques includes the following steps:
1. Requirements to be verified are represented as a finite hierarchy of properties of the system under verification.
2. For each (leaf) property **R** of the hierarchy
2.1. abstraction ($\mathbb{A}$, γ) chosen so that the target property **R** has a precise representation in the abstraction $\mathbf{A_R} \in \mathbb{A}$;
2.2. analysis done to demonstrate that $\mathbf{A_R}$ correctly represents the target property **R**;
2.3. abstract behavioural model of operating system chosen $\mathbf{A_{OS}} \in \mathbb{A}$;
2.4. analysis done to demonstrate that the chosen model $\mathbf{A_{OS}}$ adequately represents the actual behaviour of combination of operating system and hardware with respect to the target property **R** (e.g., $\alpha(\{\mathbf{HWTraces_{OSImage}}\}) \sqsubseteq_{\mathbf{HClass}} \mathbf{A_{OS}}$);
2.5. formal proof of the relation $\mathbf{A_{OS}} \sqsubseteq \mathbf{A_R}$ done using one or another verification method;
2.6. any steps of 2.1-2.5 can produce a conditional result under some assumptions or limitations that should be added to the hierarchy (step 1) for separate analysis.

4. CONCLUSION

Contributions of the paper include a systematic approach to formalization and reasoning about verification of behavioural properties of operating systems and a generic definition of conditions sufficient for sound simplification of behavioural models for verification of properties of a particular class.

The proposed approach provides a ground to explicitly define assumptions made and to formally reason about them. It is not so specific for operating systems and can be useful to any complex system under verification. But the operating systems are the most demanding application domain since accurate and comprehensive verification of operating systems requires a systematic combination of various verification techniques for checking different properties.

Applicability of the approach is limited by behavioural properties that can be expressed in collective event trace semantics. For example, probabilistic requirements are not covered by the given model. But the approach can be extended in future to cover them as well.

Another future direction is the development of a case study demonstrating how the approach can be applied to verification of a particular operating system, for example, for combining deductive verification of generic kernel code and proof of validity of assumptions made in memory model implemented in verification tools [14].

5. ACKNOWLEDGEMENT

This study was supported by RFBR grant #17-07-00734.

REFERENCES

[1] E. Cohen, W. Paul, S. Schmaltz. Theory of multi core hypervisor verification. In Proceedings of the 39th Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM'13, Berlin, Heidelberg, 2013.
[2] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, M. Norrish. Mind the gap: A verification framework for low-level C. In S. Berghofer, T. Nipkow, C. Urban, M. Wenzel, editors, Proc. TPHOLs'09, volume 5674. Springer, 2009.
[3] C. Baumann, T. Bormer, H. Blasum, S. Tverdyshev. Proving memory separation in a microkernel by code level verification. In Proc. AMICS/ISORC, 2011.
[4] C. Baumann, B. Beckert, H. Blasum, T. Bormer. Ingredients of Operating System Correctness. In Proc. embedded world 2010.
[5] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, G. Heiser. Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems, Volume 32, Number 1, pp. 2:1-2:70, February, 2014.
[6] C.A. Petri. Fundamentals of a theory of asynchronous information flow. In Information Processing 1962, Proceedings of the IFIP Congress 62, pages 386–390, Munich, Germany, 1962. North Holland Publishing Company.
[7] G. Boudol, I. Castellani. Flow models of distributed computations: Three equivalent semantics for CCS. Information and Computation, 114:247–314, 1994.
[8] G. Winskel. An introduction to event structures. In Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency, LNCS 354, pages 364–397. Springer-Verlag, 1988.
[9] G. Boudol, I. Castellani. On the semantics of concurrency: partial orders and transition systems. Proceedings TAPSOFT-87. LNCS 249, pages 123–137. Springer-Verlag, 1987.
[10] R. Milner. A Calculus of Communicating Systems. LNCS 92. Springer-Verlag, 1980.
[11] C.A.R. Hoare. Communicating sequential processes. Communications of the ACM, 21(8):666–677, 1978.
[12] J.A. Bergstra, J.W. Klop. Algebra for communicating processes with abstraction. Journal of Theoretical Computer Science, 37:77–121, 1985.
[13] P. Cousot, R. Cousot. Systematic design of program analysis frameworks. In Proceedings of the 6th POPL (San Antonio, TX), ACM Press, New York, 269–282.
[14] M. Mandrykin, A. Khoroshilov. A Memory Model for Deductively Verifying Linux Kernel Modules. In Proc. PSI-2017.