Masterthesis

Constructing a Formal Framework for Modeling and Verifying a Real Operating System

Eyad Alkassar

June 29, 2005

Saarland University, Computer Science Department
Institute for Computer Architecture and Parallel Computing
Prof. Dr. W. J. Paul

Abstract

We show how to construct a formal model of concurrently executed and communicating applications in an operating system environment. We will identify the necessary steps for building and linking abstract models of a processor, a micro kernel, and a user level operating system. The result is the outline of a formal framework that allows to prove the pervasive correctness of applications running on top of the operating system.

Eidesstattliche Erklärung

Hiermit erkläre ich, Eyad Alkassar, an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, im Juni 2005

Danksagung

Mein Dank gilt Prof. Paul, der mir dieses spannende Thema angeboten hat und meine Betreuung übernahm. An dieser Stelle möchte ich allen Mitarbeitern des Lehrstuhls, insbesondere Sebastian Bogan für die wertvollen Anregungen und Diskussionen danken. Besonderer Dank geht an meine Eltern und an meine Geschwister Ammar, Muhannad und Manar für die grenzenlose Unterstützung und Ermutigung.

Contents

1 Introduction

In today's world we rely in nearly all security-relevant spheres of our daily life on computer systems. From simple circuits in traffic lights to complex control software of nuclear power plants we *believe* in the infallibility of computers. But who or what ensures that these systems really do what they are intended to do? One way would be to *test* all possible cases, or at least a set of representative ones. With a complexity of programs increasing in a exponential way, this approach is futile if one wants to ensure that *no* "bugs" are left undetected.

The alternative to testing is *proving* correctness in a mathematical sense, by first formulating *accurate models* of the real-world system and then verifying formal assertions over these models. This approach was pioneered by, among others, Dijkstra, Flyod, Lamport and Hoare.

But when is a model *accurate*? When does a model capture all relevant information of a real system? To date most approaches in formal verification only model some limited parts or high-level layers of an overall system. Underlying layers are either modeled in some semi-formal way or some *perfect* (i.e. correct) and simplified model is even postulated, claiming an accurate model would be too complex to build. For example when a concrete application, as a fingerprint authentication software, running on top of a real operating system, as *Linux*, should be verified, it is not enough only to prove the correctness of the abstract fingerprint matching algorithm. This is because the specification of this algorithm may use some model of the services provided by the operating system, *assuming* both correctness of modeling and of functionality of the services.

In [20] J. S. Moore, principal researcher of the CLI stack project [3], declares the formal verification of a "practical computing system from transistors to software" as a grand challenge problem.

A main goal of the Verisoft project [7] is to bear this challenge. In the academic system, a subproject of the Verisoft project, a general-purpose computer system, covering all layers from the gate-level hardware description to communicating concurrent programs is designed, implemented and verified. The aim is to build accurate formal models without hidden assumptions, that are modular and easily extendable. The verification should be pervasive throughout all layers of abstraction and take advantage of computer-aided verification tools.

Joining all developed models into one consistent and pervasive framework and sketching the most important correctness properties is the goal of this thesis. A *sequence* of models is constructed and techniques for formally relating these models with each other are described. For this we will use the two main techniques *instantiation* and *abstraction*.

In fine: The framework we want to design should be *complex enough* to face real world verification challenges and *simple enough* to be taught in an undergraduate computer science course.

2 Terminology

2.1 About types

Types are sets of mathematical objects. We will use two sorts of types, basic types and composed types. Basic types that we use are *booleans* (notation is *bool*), *natural numbers* (*nat*), *integers* (*int*) and *strings*. We will distinguish five kinds of composed types: *sets*, *lists*, *arrays*, *records*, and *functions*:

- **Sets**
  A set type is defined over a basic or composed type *t_typename*. Elements of a set type are sets of elements of the specified type. The notation for set types is *t_set_typename*. We us the standard mathematical notation for set operations.
- **Lists**
  A list type is defined over a basic or composed type *t_typename*. Elements of a list type are ordered lists of elements of the specified type. The notation for list types is *t_list_typename*. The first component of a list *l* is accessed by *head(l)* and the rest of the list by *tail(l)*. An arbitrary component *i* is accessed by *l[i]*. Two lists *l1, l2* are concatenated by *l1 @ l2*. The empty list is denoted by *nil*.
- **Arrays**
  An array type is defined over a basic or composed type. Elements of an array type are written as $[a_1, a_2, \ldots, a_n]$. The notation for array types is *t_array_typename*. Component *i* of an array *a* can be accessed by *a[i]*.
- **Records**
  Record types are written as
  t_typename = *(label*$_1$*: type*$_1$*, label*$_2$*: type*$_2$*,..., label*$_n$*: type*$_n$*)*. Elements of record types have the form *(label*$_1$ = *value*$_1$*, label*$_2$ = *value*$_2$*,..., label*$_n$ = *value*$_n$*)*. Component *x* of record *r* is accessed by *r.x*.
- **Functions**
  A function type is defined over two basic or composed types. Elements of a function type are sets of pairs.
  The notation for functional types is *t_typename1* $\longmapsto$ *t_typename2*.

Values of all types can be assigned by the $:=$ operator. In the following we will use finite-state machines. Their state definition will be given through record types.

2.2 What is a model?

Models are used in a widespread range in scientific literature: models as theories of nature, models as abstract mathematical theories or just models as specifications. Intuitively all these interpretations follow the idea of formally catching relevant information and behavior of some real system or of another model. For our approach we will use finite-state machines as formal language for modeling.

A model M is defined as a quadruple consisting of the four parts:

Definition Model: A model M is a quadruple $\langle S, I, input, trans \rangle$ with $I \subseteq S$ and $trans \subseteq S \times (input \cup \{\tau\}) \times S$

- **state space** S The state space defines all valid states of our model. A state in the model will often be a set of states of some kind of processes. Formally we will describe state spaces as *record types*.

- **init space** I The init space is a subset of the state space and defines all valid start states of the model.

- **input signals** $input$ Each model has a set of input signals. The input signals are occurring in a nondeterministic and asynchronous way. Mainly input signals will be interrupt events.

- **transition relation** $trans$ The transition function *trans* takes a state and an input signal or the empty word τ and returns the corresponding set of states reached by the finite-state machine in the next step. The transition relation itself often consists of a set of transition relations defined on components of the state space.

M.x denotes the component *x* of a model *M*. As convention we will use capital letters M,N for models and A,B,C for states of a model. The small letter a will be used as meta variable for input names. For convenience we will write A $\xrightarrow{a}$ B if $B \in trans(A, a)$ and A $\xrightarrow{\tau}$ B if $B \in trans(A, \tau)$. The latter ones are called *internal* transitions. Internal transitions can be annotated by names. For example if the internal transition was the execution of a task with ID x, we will write $\xrightarrow{\tau_{task\_x}}$.

For expressing the execution of arbitrary many internal events under one input signal we write A $\stackrel{a}{\Longrightarrow}$ B $\stackrel{def}{\Leftrightarrow}$ A $(\xrightarrow{\tau}{}^{*} \circ \xrightarrow{a} \circ \xrightarrow{\tau}{}^{*})$ B. For expressing the execution of $k \geq 0$ internal transitions we write A $\xrightarrow{\tau}{}^{k}$ B.

In the following we give the definition of *runs of a model*:

Definition Runs of a model: A pair $(\sigma, \alpha) : t\_list\_M.S \times t\_list\_M.input$ is called run of the model M iff $\sigma[0] \in M.I \wedge \forall i : nat\ \sigma[i] \xrightarrow{\alpha[i]} \sigma[i+1]$.

Throughout this thesis we will construct a sequence of models. Every model must be related in some sense to its predecessor. We will identify two main techniques for building new models: *instantiation* and *abstraction*.

2.3 Instantiation

The first method for constructing new models is instantiation. Every model has a set of parameters (e.g. the set of applications, drivers, the concrete operating system, etc.). By choosing a certain value (or restricting the type) for one or more parameters a new model is introduced which we will call an instance of the previous one.

Definition Instantiation: A model M is called instance of a model N, if state space and init space of M are subsets of state space and init space of N and all other components of M and N are equal.

It is not directly obvious why this definition fits our intuitive notion of instantiation. Especially what are *parameters* in our formal framework? According to our definitions one model can contain many finite state machines. Every single machine is defined through one single start state. Therefore restricting parts of init space to smaller ranges (or to certain values) can be considered as instantiation and vice versa.

Formally every variable for which the cardinality of the set of occurring values in the state space is greater than one is a *parameter* .

If a component *x* of the state space of Model *M* is instantiated to a concrete value *c* we will write $M[x := c]$.

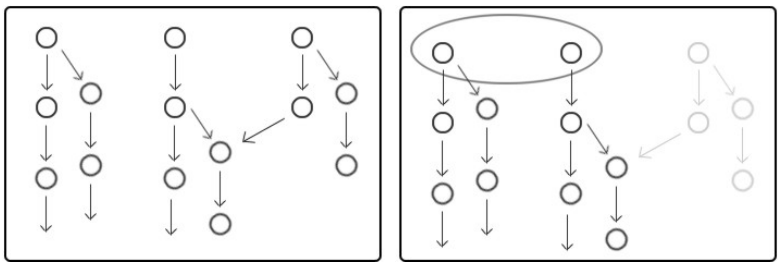

Figure 1: Instantiation through subsets of init space

2.4 Abstraction

In science the term *abstraction* is by far not used unambiguously. Physical laws *abstract* natural processes, programming languages *abstract* actual execution of assembler code and so on. In a pure mathematical interpretation, abstractions can be considered as base changes between two algebras. The Fourier transform, for example takes a signal in spatial space and transforms it into frequency space, in which (some types of) periodicities are better visible. In many cases abstractions are also used to compress information (either the state space of the model or the length of runs).[1]

For our approach, abstractions should serve two goals:

- **Downwards** the abstraction is a specification of the underlying concrete system. In most cases the concrete system refers to some code implementation (higher programming language, assembler, etc.).

- **Upwards** the abstraction represents a compressed and easily accessible representation of functionalities for verification purposes and for the design of the next higher model.

Obviously the abstract and the concrete model must be linked in a formal way, or in other words we must define when a model is an abstraction of another one. This definition is crucial, because it says what type of information loss is allowed and with that how properties propagate between two models.

2.4.1 Correctness of abstract models

In order to formally describe when a model is an abstraction of another one we have chosen the *implements* relation defined below. It says whenever the abstract model performs some computation under a certain input signal, the concrete model is able to *simulate* the step under the same or some equivalent input but with arbitrary many internal transitions (i.e. also zero transitions are allowed). The implements relation is similar to the *weak simulation* notion of Robin Milner [26].

[1] Jpeg-Compression is nothing but a base change into the wavlet space with a dimension smaller than the original spatial space.

Definition implements: Given two models M and N and a simulation relation SIM $\subset M.S \times N.S$. We say model N implements model M under SIM, iff

- $\forall A \in M.I \; \exists C \in N.S$ (A, C) $\in$ SIM
- whenever (A,C) $\in$ SIM it holds
  if $A \xrightarrow{\tau} A'$ *then* $\exists C' \in N.S \; C \xrightarrow{\tau}{}^{*} C' \wedge (A', C') \in SIM$
  if $A \xrightarrow{a} A'$ *then* $\exists C' \in N.S \; C \xRightarrow{a'} C' \wedge (A', C') \in SIM$

The first condition ensures, that all valid start states of the abstract model can be mapped to some concrete ones. In the second condition the input signals a and a' are related by some translation function. The implements relation can be understood as *correctness* of functionality of the abstract model: Every execution in the abstract model is also one in the concrete (or implemented) model. It ensures the propagation of functionality properties (we will not define properties formally) from the abstract level *down* to the concrete one.[2]

Unfortunately this is *not* enough for our purpose. Our aim is to design an abstract model of an operating system in which not only correctness of specified functionality is verified but also *security* properties. Security properties state the *absence* of bad events. Bad events are for example a crash of the system (caused through malicious functionality) or prohibited information flow. If these security properties should be verified at an abstract level we must ensure that these bad events *propagate* correctly *upwards* from the lowest level.

2.4.2 Completeness of abstract models

The necessity of completeness becomes clear, when considering programming language abstraction. At some layer in our framework we could represent programs as a list of assembler instructions where in the abstract model only C-Programs are considered. The implements relation would hold (under the assumption of compiler correctness). However security properties formulated at the abstract layer, like the assertion *no program can cause the kernel to crash*, are fruitless. That is because on assembler layer there could be some malicious constructs that can not be generated by the compiler (indeed this is one major intent of compilers, namely to restrict programs to *safe* operations by prohibiting pointer arithmetic for example).

There are many ways to tackle completeness. The simplest one would be to state all security properties only at the lowest level of abstraction. This solution would make all benefits of abstraction obsolete. Most of the security properties we want to formulate become only visible in abstract layers. For example the assertion over the integrity of data of user tasks (see Section 5.4.2) is senseless at processor layer where the term user process is not even defined.

A more constructive solution to avoid completeness problems is to build the framework *top-down*. Starting with the highest level of specification, step by step only those functionalities would be implemented that are specified. This is exactly the way the CLI-stack [3] tried to handle the issue of completeness. This procedure has two main drawbacks. Firstly, it is goal-oriented more than analytic. That means we would not

[2] *Implements* is always proven in respect to the simulation relation. Therefore SIM must be chosen in a correct way.

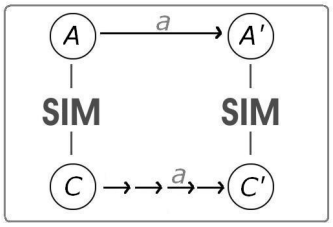

Figure 2: Implements relation

try to build models for real systems. We rather specify how systems should look like in every layer considering all assumptions of the top level. In fact this is unproblematic until it comes to hardware. Here we have some concrete and verified architecture on which we want to build our stack. In the CLI-stack this problem is obsolete only because the stack is not built upon a concrete hardware. Rather some Von-Neumann architecture is specified.
Secondly, this approach gives no general and formal technique to check for completeness. It rather helps to avoid that functionality is present in the implementation but not at the abstract layer.

In another, related approach all security properties are identified at the highest level of abstraction and then some needed, general security guidelines, e.g. information flow policies, are (translated and) verified for *each* model. In fact this would be a satisfying solution.

The most general solution is to define a relation, similar to the implements relation that states when a model is *complete* in our sense. That means when this completeness relation holds certain types of security properties will propagate upwards.

In the following the definition of *equivalence* between two models is given. *Equivalence* states both completeness and correctness of the abstraction.

Definition equivalence of models: Given two models M and N and a simulation relation SIM $\subset M.S \times N.S$. We say model N and model M are *equivalent* or complete and correct, iff N implements M under SIM and M implements N under SIM^{-1}.

One could argue that with both completeness and correctness abstraction becomes senseless. This is not the case. Our notion of equivalence includes information loss between models (see temporary abstraction below). As explained above also abstractions without information loss (even without any data compression) can be extremely helpful in understanding certain aspects of a model.

2.4.3 Categories of abstraction

We identify four types of abstractions: *data*, *functional*, *structural* and *temporal* abstraction (This categorization is mentioned also in [25] and is not formally described). Indeed not every case of all types of abstraction fits our requirements, namely correctness *and* completeness. As explained above one has to be extremely careful not to *abstract* error cases *away* when choosing a model.

In the following every type of abstraction is briefly sketched and possibly problematic cases are illustrated.

- **Data**
  Data abstraction is a well studied subject in program verification (for details see Chapter 24 in [24]). We will talk of data abstraction whenever a data structure in the concrete model is represented through a more accessible (for our purposes) one in the abstract model.

  In some low level models, instructions of the hardware machine could be treated as hexadecimal numbers. A more abstract model would represent instructions as some set of names. If this set of names is smaller than the space of used hexadecimal numbers we obviously lose information. The new abstraction than either merged or even ignored possible instructions. Although correctness (of the modeled set of instruction) could still hold for the abstract model, some (possibly malicious) functionality could be ignored and therefore completeness would be violated.

- **Functional**
  Given a procedure in some programming language, a functional abstraction may describe its semantic effects (in some other formal language). Functional abstraction is often used together with data abstraction. If the procedure is executed sequentially and non-interleaved in the concrete model, and signatures (i.e. the type) are equal in both concrete and abstract model, usually a proof of equivalence is straightforward (e.g. by using Hoare-Logic).

  Problematic cases arise with interleaved executions. A functional abstraction may consider the execution of the concurrently executed procedure as a single atomic step.

- **Structural**
  Structural abstraction refers mainly to communication mechanisms. In most cases it is applied in combination with functional and data abstractions.

  Example: The bus communication protocol between the operating system and the hard disk. On implementation side, a read-file operation from hard disk may take several steps (sending the read request, waiting for an answer, requesting the first 100 bytes of the file wait for an answer,...) which may even be interleaved with the execution of other applications. On abstraction side, the whole mechanism is represented by a single *atomic* primitive.

- **Temporal**
  The notion of time is treated in different ways in our framework. From model to model coarser steps in time are considered. Starting with real time on the real physical machine (or upper bounds for cycle times on first models of hardware abstraction), later on only the order of execution is coded. By doing so information is *lost*. However this loss of information is not visible under our concept of equivalence.

  A different case is if the abstract model even ignores the concrete order of execution. Such approaches may conflict with our definition of correct and complete abstraction.

This work does *not* claim to give a complete formalization of the framework. In fact terms as property, equivalence, simulation and abstraction have to be subject of a much more detailed study. Furthermore a complete formalization in some well known calculus as *CCS* (for more information see [12]) or in form of *I/O-Automata* (see [19]) could be desirable. In the following we will use a graphical representation (which is quite similar to communication graphs in CCS) of models to emphasize important communication structures.

2.5 Constructing the simulation relation

The problem of showing equivalence is to find a suitable simulation relation SIM. Especially when proving completeness, most states in the concrete model have to be mapped to abstract states.

Given that a model N implements a model M under a relation SIM, in some cases one can generate automatically a relation SIM' similar to SIM, s.t. N and M are equivalent under SIM'.

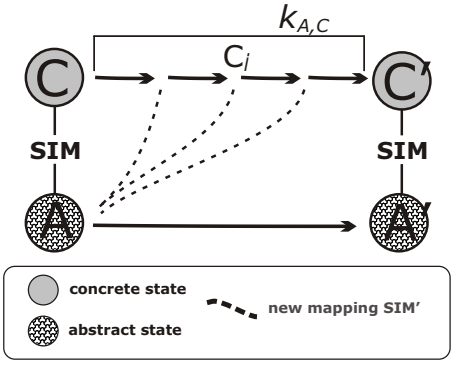

Figure 3: Constructing the new mapping

One such case is when both models M and N are deterministic and when each concrete start state can be mapped to an abstract one. If furthermore only models without input signals are considered, one easily can see how each concrete run can be mapped to an abstract one.

With that we get the following four conditions that have to hold for constructing the simulation relation SIM'.

- **C1** N implements M under SIM.
- **C2** The models N and M are deterministic, i.e. $N.\delta$ and $M.\delta$ are functional.
- **C3** N and M have only internal transitions.
- **C4** Every concrete start state can be mapped to an abstract one: $\forall C \in N.I$ $\exists A \in M.I$ (A, C) $\in SIM$

For convenience we define a predicate *CondDet(N, M, SIM)* to be true, iff the conditions C1 to C4 hold for N, M and SIM.

Now we can define the new Simulation relation SIM'. The idea is very simple. Given some abstract transition A $\xrightarrow{\tau}$ A' and the corresponding sequence of concrete states starting at C and ending in C', we map every concrete state between C and C' to A (see figure 3).

Let $k_{A,C}$ denote the number (greater zero) of internal transitions that have to be taken from C until the first state C' is reached that maps to the τ-successor of A (i.e. $N.\delta(A,\tau)$, see figure 3). We know from C1 and C2 that for each $(A,C) \in SIM$ exactly one such number $k_{A,C}$ exists.

$$SIM' = \{(C_i, A) \mid \exists (C,A) \in SIM^{-1}.\ \exists 0 \leq i < k_{A,C}.\ C \xrightarrow{\tau}{}^{i} C_i\}$$

It is easy to see that $SIM^{-1} \subseteq SIM'$ holds (set i to zero). Now we prove that the definition of SIM' really suffices for proving equivalence.

Lemma For all models M, N and all relations SIM:
$CondDet(M,N,SIM) \implies$ N and M are equivalent under SIM'.

Proof We have to prove that N implements M under SIM'^{-1} and that M implements N under SIM'.

- **N implements M under** SIM'^{-1}
  The first condition of the implements relation is directly satisfied through C1: $\forall A \in M.I\ \exists C \in N.S$ (A, C) $\in SIM$ and because $SIM^{-1} \subseteq SIM'$.

  For proving the second condition we assume $(A,C) \in SIM'^{-1} \wedge A \xrightarrow{\tau} A'$. By the definition of SIM' and k_{A,C_0} we know that $(A,C) \in SIM'^{-1} \implies \exists (C_0, A) \in SIM^{-1}.\ \exists 0 \leq i < k_{A,C_0}.\ C_0 \xrightarrow{\tau}{}^{i} C \xrightarrow{\tau}{}^{k_{A,C_0}-i} C' \wedge (A', C') \in SIM'^{-1}$.

- **M implements N under** SIM'
  The first condition of the implements relation is directly satisfied through C4: $\forall C \in N.I\ \exists A \in M.I$ (A, C) $\in SIM$ and because $SIM^{-1} \subseteq SIM'$.

  For proving the second condition we assume $(C,A) \in SIM' \wedge C \xrightarrow{\tau} C'$. By the definition of SIM' we know that $\exists (C_0, A) \in SIM^{-1}.\ \exists 0 \leq i < k_{A,C_0}.\ C_0 \xrightarrow{\tau}{}^{i} C$. We distinguish two cases.

  First case: $i < k_{A,C_0} - 1$. In this case we know that the direct successor of C, C' is also mapped in SIM' to the abstract state A. This is because for (A, C') it holds that $C_0 \xrightarrow{\tau}{}^{i+1} C'$ and $i + 1 < k_{A,C_0}$.

  Second case: $i = k_{A,C_0} - 1$. By the definition of k_{A,C_0} we know that $C_0 \xrightarrow{\tau}{}^{k_{A,C_0}} C'$ with $(M.\delta(A,\tau), C') \in SIM$. Because $SIM^{-1} \subseteq SIM'$ it follows that $(C', M.\delta(A,\tau)) \in SIM'$.

q.e.d.

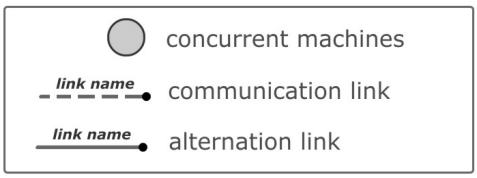

Figure 4: Elements of the graphical representation

2.6 The programming language C0

The main programming language used in the *Verisoft* project is *C0*. *C0* is a imperative and high level programming language whose syntax is similar to C and whose semantics is unambiguously defined. A detailed description of the small step semantics can be found in [2]. C0 is the choice programming language because a verified C0 compiler exists [17]. To verify C0 programs against some specification the theorem proving environment *Isabelle* (for details see [22]) is used in Verisoft.

2.7 Graphical Representation

As described above states of models consist of many parts. Some of them are states of machines (processes, kernel, devices, etc.) that are concurrently executed in the model. Such machines will be represented by circles (see figure 4). These machines have their own transition relation (part of the model transition relation) which manipulates only their local state. Parts of the transition relation that manipulates the global state of the model are represented in the figures in form of *links*. Global transitions which represent communication (as send or receive statements in programs, shared variables, etc.) are the so called *communication links*. Other global transitions invoked by machines or devices are called *alternation links*.[3]

[3] Our graphical links are exactly not-bound names in CCS or external actions in the I/O-Automaton.

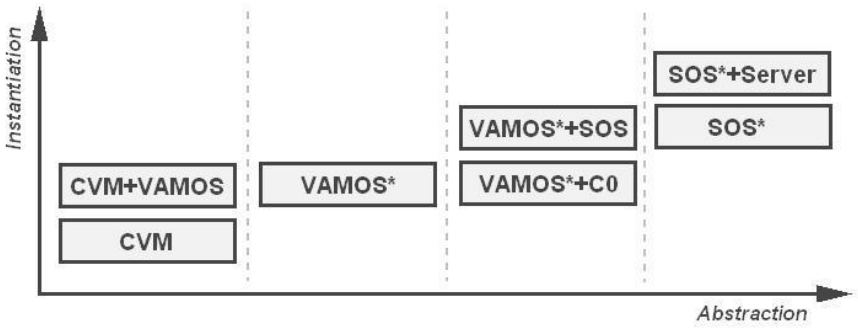

Figure 5: The verification stairs

3 Course of action

Our goal is to design and verify a top level model of communicating applications in operating system environment. At the same time the ambition of Verisoft is to have pervasive verification starting from the processor.

All intermediate layers of the system must be described and modeled. We identify three main layers of construction: processor, micro kernel and operating system. All layers are real systems not only specifications of how the system should look like. The used hardware is a pipelined, DLX Machine [21]. On top of it a micro kernel called VAMOS is established. The last layer is given through the Simple Operating System (SOS) built on top of VAMOS.

This thesis is a road map for modeling and verifying these layers. A *sequence of models* is constructed and should be later on verified. For constructing new models we will use the techniques presented in Section 2 namely *instantiation* and *abstraction*. The following scheme is applied at least once to each of the mentioned layers:

- Instantiating the model of the underlying layer.
- Abstracting the main functionalities of the instantiated model (we gain a specification of the former instantiated model).
- Show how equivalence between instantiation and abstraction could be proven and state important properties of the abstraction.

This scheme is illustrated in the *verification stairs*that we will ascend (see figure 5). For each model we will describe the state space in detail and sketch the transition relation without stating the exact semantic of each introduced primitive.

Each model has to be concrete enough to catch all necessary behavior of applications and abstract enough to easily formulate assertions and correctness properties. In fact we are always searching the simplest model of a layer that is expressive enough for the properties we want to prove. Furthermore our models should always be easily extendable in both, new functionalities and their proofs.

Our starting point is an abstraction of the used processor, called the VAMP (for details see [4]). This model is specialized (by instantiating a special DLX-Code) and abstracted (by only considering the effects of this code) to the model *Communicating Virtual Machines* (CVM*, see Section 4). The next modeled layer is the kernel layer.

The layer is formalized by instantiating CVM* with the VAMOS kernel code and abstracted to the model VAMOS* (described in Section 5). So far we only considered processes as Assembler instruction lists. Obviously it is preferable to argue about and verify higher level programming languages, as C or Java. The model VAMOS*+C0 (see Section 6) introduces the semantics of the high-level programming language C0 (the syntax of C0 is unambiguously defined in [16]). In the next step one C0 application is instantiated to the *Simple Operating System*. This instantiation is finally abstracted to the desired top-level model SOS* (Section 7).

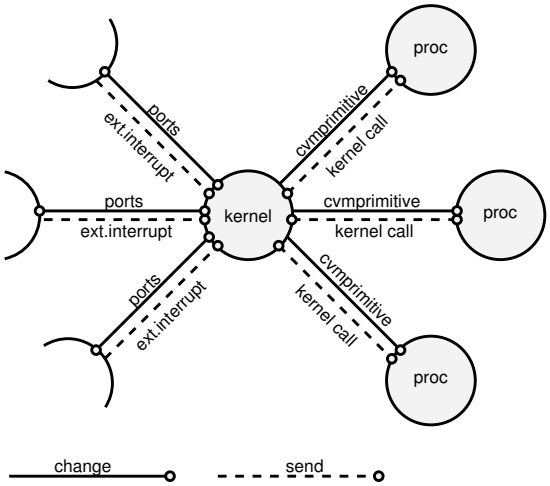

Figure 6: Communication structure of CVM*

4 Communicating Virtual Machines

4.1 The model

Communicating Virtual Machines [9] is our first model on top of the processor layer. CVM* simulates the execution of different DLX Assembler programs as if they were executed on separate machines with own memory and registers. With that the notion of separate processes is introduced. One particular process (the kernel) organizes the *interprocess communication (IPC)* and access to resources and devices. These services are provided to other processes via interrupt mechanisms.

The model CVM* is given through:

Model definition: Communicating Virtual Machines

$$CVM* = (t\_cvmstate, t\_cvminput, t\_cvmstart, \delta_{cvm})$$

One input signal of CVM* is a list containing either different external interrupt signals or external port changes (see devices). In the following we will describe $t\_cvmstate$ and δ_{cvm} in detail.

4.2 State space of CVM*

The CVM* model identifies two types of processes: *kernel* and *user processes*. During each execution step exactly one of these processes is active, the *current process* (cp). CVM* is communicating with external devices via so called ports. The state of these ports is described in the component *devices*. The location of the handlers for interrupts are specified in *handler*. Hence, we get a record consisting of five parts for the CVM* state.

t_cvmstate = (kernel : *t_c0machine*, procs : *t_vm*, cp : *nat*,
devices: *t_list_device*, handler: *t_handler*)

4.2.1 Kernel

The CVM* model follows a μ-kernel architecture. In contrast to monolithic kernels as well-known operating systems Windows or Linux, μ-kernels only provide processes with a minimum of service, mainly communication (IPC) and memory management (in [18] μ-kernel pioneer Liedtke describes the most important services that have to be provided). Therefore μ-kernels lead to a more modular system, which is exactly what we want for verification and modeling.

In CVM* the kernel is modeled as a C0-machine. A C0-machines contains a C0 *program*, which is a statement list. Possible statements are either usual C0 commands or the so called *CVM* primitives*. We need *CVM* primitives* for operations changing the global state of the CVM* model as creating new processes, memory allocation, etc. (*CVM* primitives* are implemented as inline assembler code). Further the kernel has a table containing all needed type information, a *local memory* (consisting of a heap and a stack), a *function table*, containing the signature and code of all callable functions, and *services* which is also a function table containing signature and code of all services provided by the kernel to the user processes (types of the components are omitted, for details see [2]):

t_c0machine = (program, typetable, memory, function_table, services)

In fact the kernel code can be separated in three main parts: code for handling the timer interrupt (or the scheduler), code for handling external interrupts other than the timer and code for handling the trap interrupt invoked through one of the user processes.

4.2.2 User Processes

The maximum number of user processes *pn* is constant. The process component of the CVM* model state is given through:

t_vm = (pn: *nat*, processes: *t_list_process*)

In contrast to the kernel, user processes are modeled as Virtual DLX Assembler machines (for details see [21]). Therefore each user process consists of seven components:

- **Process ID (PID)** Each process has an unchangeable and unique ID.
- **General Purpose Register (gpr)** The *gpr* is a set of 32 register cells. Each register is 32 bit long.
- **Special Purpose Register (spr)** When a local interrupt occurs some needed information (interrupt type, etc.) are saved in these registers.
- **Program Counters (pc and dpc)** The program counters indicate the next assembler instruction that should be executed.
- **Instruction List (program)** *program* is a list of assembler instructions that represents the program of a process.
- **Main Memory (mm)** *Main Memory* is a mapping between addresses in the virtual space of the process and content of the memory cells.

With that we get for the type of one process

t_process = (pid: *nat*, gpr : $\{0,1\}^5 \longmapsto \{0,1\}^{32}$, spr : $\{0,1\}^5 \longmapsto \{0,1\}^{32}$, pc : *nat*, dpc: *nat*, program: *t_list_instr*, mm: $\{0,1\}^{32} \longmapsto \{0,1\}^{32}$)

4.2.3 Devices

Our model implements so-called *memory-mapped I/O* (*mmIO*, for details see [13]). In *mmIO* communication between kernel and devices is realized over changes of certain memory cells (some set of cells for each device) called *ports*. In CVM* devices are not explicitly modeled as finite-state machines. Rather they are treated as source of input (namely changes to the ports and occuring of interrupts).

One device is given through the following record in the CVM* model:

t_device = (pc : *nat*, content : $\{0,..,pc-1\} \longmapsto \{0,1\}^{32}$, active : *bool*)

- **Port count (pc)** Every device has a certain number *pc* of ports, that can be written and read by the kernel.
- **Content** This variable stores the current content of one port of the device. A port could be considered as a memory cell, that represents a part of the interface between kernel and I/O device.
- **Active flag** An active flag with the value *false* indicates that the device is currently not allowed to manipulate its ports. In this case normal memory semantics hold for ports.

4.2.4 Interrupts

There are two main categories of interrupts, *internal* and *external* interrupts. External interrupts are caused by I/O-devices. External interrupts are indicated via the input signals. The CVM* model specifies that they are handled by kernel code. Internal interrupts can be separated into the types *traps* and local interrupts. Local interrupts have only influence on the process which caused the interrupt. For example *arithmetic overflow* or *division with zero* are such interrupt types.

In contrast the *trap* interrupt has global effect. In fact it is the basic communication primitive of processes. It is caused by the *trap-instruction* and causes the kernel to execute one of the specified services (the most important one is IPC).

The mapping between interrupt type and handler is given through the last component of the cvm_state, *handler*.

t_handler = (iohandler : $nat \longmapsto nat$, traphandler : $nat \longmapsto nat$, localhandler: $nat \longmapsto nat$)

- **Handler for I/O interrupts** This function takes the level of the occurred interrupt and returns the number of a kernel function assigned to it (for example the device driver).

- **Handler for traps** When the trap-instruction is called also an immediate constant is delivered. The trap handler assigns to each such immediate constant the number of the specified service in the kernel data structure *services*.

- **Local handlers** This function assigns to the interrupt level a position in the instruction list of the process for handling the interrupt.

Having specified the state of the CVM* model, we now can define its semantics through the transition relation δ_{cvm}.

4.3 Semantics of CVM*

The transition relation of the CVM* model is functional. It takes one state of the CVM* model and returns the next reached state:

CVM* transitions $\delta_{cvm} : t\_cvmstate \times t\_cvminput \longmapsto t\_cvmstate$

As mentioned in Section 2 the transition relation is composed of the transition relations of all machines of a model. In the CVM* model δ_{cvm} consists of four main parts: transition of a C0-machine, execution of a CVM*-primitive, transition of a Virtual DLX Assembler machine and interrupt handling.

- **Transition of a C0-machine**

  $\delta_{c0} : t\_c0machine \longmapsto t\_c0machine$

  This transition function takes the state of a C0-Machine *c* and executes the first ordinary C0-statement in *c.program*.

- **Execution of a CVM*-Primitive**

  $\delta_{prim} : t\_c0machine \longmapsto t\_c0machine$

  This function takes a CVM* state *cs* and applies to it the semantics of the CVM* primitive which is the first statement of *c.program*.

- **Transition of a DLX Machine**

  $\delta_{dlx} : t\_process \longmapsto t\_process$

  This function executes the instruction in position *p.dpc* of the given process *p* .

- **Interrupt handling**

  The function *isInterrupt* indicates that either a local or an external interrupt or a port change occurred during the last transition:

  $isInterrupt : t\_cvminput \times t\_cvmstate \longmapsto bool$

  All interrupts are handled by:

  $\delta_{interrupt} : t\_cvminput \times t\_cvmstate \longmapsto t\_cvmstate$

  Given an input *i* and a state *cs* this function updates the port changes reported in *i* and executes the handlers of the reported external interrupt. In the case an internal interrupt has occurred, it starts the local interrupt handler. Given more than one interrupt the order of executed handlers is chosen deterministically.

With the help of these four transitions we can define δ_{cvm}. If the current active process is the kernel (i.e. $cp = 0$) two cases could happen. Either the next statement is an ordinary C0-statement, then δ_{c0} is applied to it. Otherwise the next statement is a CVM* primitve. In this case δ_{prim} is applied. If an interrupt occurred (that means either the input is not empty or a local interrupt occurred in one of the processes) the adequate handler is called with $\delta_{interrupt}$. If no interrupt occurred and a user process is active δ_{dlx} is applied to it. One restriction of the occurring of interrupts is given through the hardware: during the execution of a kernel call or a handler, interrupt signals of the same type must not change.

Formally δ_{cvm} is given through:

$$\delta_{cvm}(c,i) = \begin{cases} \delta_C(c) & : \text{ if } c.cp = 0 \\ \delta_{VM}(c) & : \text{ if } c.cp > 0 \wedge \overline{isInterrupt(c,i)} \\ \delta_{interrupt}(c,i) & : \text{ if } c.cp > 0 \wedge isInterrupt(c,i) \end{cases}$$

with

$\delta_{VM}(c).procs.processes[c.cp] = \delta_{dlx}(c.procs.processes[c.cp])$

and

$$\delta_C(c).kernel = \begin{cases} \delta_{c0}(\delta_{prim}(c).kernel) & : \text{ if } c.kernel.program = CVMPrimitiv; rest \\ \delta_{c0}(c.kernel) & : \text{ if } c.kernel.program = \overline{CVMPrimitiv}; rest \end{cases}$$

4.4 Correctness of CVM*

The CVM* model itself is implemented on an abstract processor model, the VAMP [4]. The implementation covers code for memory management, the CVM* primitives and the page fault handlers. We will call this model, VAMP instantiated with CVM* code, VAMP+CVM [8]. The correctness of the CVM* model is than given by a relation CSIM and the following theorem:

Theorem 1 The models VAMP+CVM and CVM* are equivalent under the relation CSIM.

The most important abstraction and therefore the main part of the proof of equivalence is the representation of DLX programs (i.e. some instruction list) as a virtual process with own, virtual memory and registers. This logical isolation of processes is described in the simulation relation CSIM and opens the door to a concurrent software model. Furthermore the CVM* model introduces C0 semantics for the kernel and the CVM* primitives. The equivalence proof therefore depends on the C0 compiler correctness and on the correct implementation of the CVM* primitives.

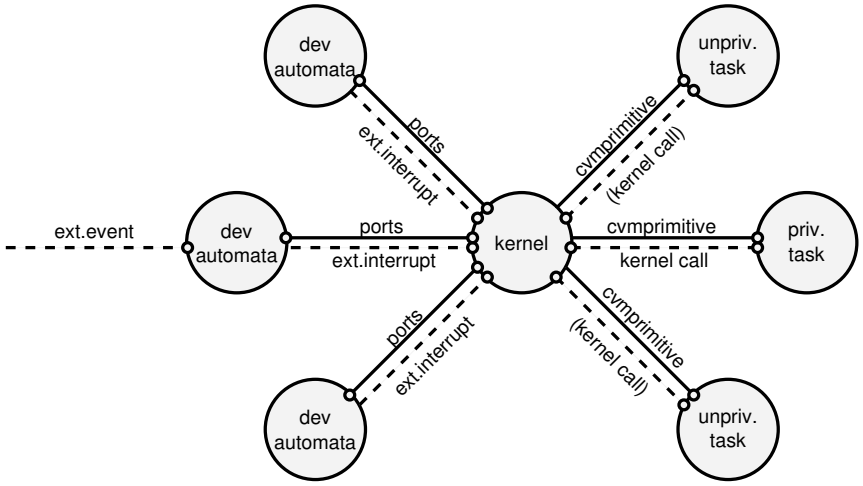

Figure 7: Communication structure of the VAMOS* model

5 Communicating Tasks

5.1 The model

After having defined CVM* we will approach the kernel layer. So far the kernel (written in C0) is only a *parameter* of the CVM* model. We will now instantiate the kernel in CVM* to one concrete C0-machine, called VAMOS-machine (for details see [10]). The new instantiation is called CVM*+VAMOS model.

As mentioned before the kernel mainly consists of code for the handler of the timer interrupt (the scheduler), code for handling other external interrupts (the drivers) and code of the services provided to the user processes. In the following we will call these services *kernel calls*.

Model definition: VAMOS Instantiation of Communicating Virtual Machines

$$\textit{CVM*+VAMOS} = \textit{CVM*[kernel:= vamos-machine]}$$

Obviously for higher layers the concrete C0 code of the kernel is not any more interesting. Rather only its semantical effects should be modeled. In the next step we will therefore introduce an abstraction of CVM*+VAMOS. The new model is called VAMOS*. In VAMOS* we introduce two main types of abstractions: no code is visible anymore for the scheduler, services nor drivers (functional abstraction). Furthermore some data structures of the IPC mechanism are modeled as part of the processes (structural abstraction) and not as part of the kernel. These extended (by IPC and some further data structures) processes are called *tasks* in VAMOS*.

The model of VAMOS* is given through:

Model definition: Communicating Tasks

$$\textit{VAMOS*} = (t\_vamosstate, t\_vamosinput, t\_vamosstart, \delta_{vamos})$$

In contrast to CVM*, devices are modeled *explicitly* as finite-state automata. Therefore most input signals of CVM* become internal transitions in VAMOS*. Only some *unpredictable* signals as a keystroke, etc. are still represented as real input to the model

(more precisely to the devices).

In the following the concrete state and the transition relation are defined.

5.2 State Space of VAMOS*

In the state space of the VAMOS* model no code of the kernel is visible anymore. Rather we model only the effects of this code and some needed data structures. Theses remaining data structures of the kernel are modeled in the component *kds*. The second component is a list of all tasks of the system. Basically a task is an extended process. At each time exactly one process or a device is active, the one with the ID given in *ct*. External interrupts (generated by transitions of the devices) that occurred in the last step and have not been handled yet are saved in the variable *ex*. The last component is a list of all devices connected to the system:

t_vamosstate = (kds : *t_kds*, tasks : *t_tasks*, ct : *nat*, ex : *t_list_exint*, devices: *t_list_vdevice*)

5.2.1 Kernel data structures (kds)

When abstracting we try only to model a minimum of needed data structures (in relation to equivalence). For modeling the kernel in VAMOS* we will only need data structures for scheduling, for time and for interprocess communication.[4]

t_kds = (readylist : *t_list_nat*, sleeplist : *t_list_nat*, time : *nat*, cupst : *nat*)

- **The ready list**

  Every task (i.e. its ID) is either in the ready list or in the sleep list. A task in the ready list is called ready. Ready tasks take a transition in the next step (i.e. they are executable) if they are scheduled. This data structure is managed by the scheduler.

- **The sleep list**

  Tasks in the sleep list are called sleeping. A task is moved from ready list to sleep list in two cases: either it called an IPC send service (*ipcSend* or *ipcSendReceive*) and neither did the intended receiver receive the message nor did the specified timeout of the kernel call expire. Or the task invoked the kernel call to receive from another task that has not yet sent a message to it.

- **Time**

  The variable *time* is not directly related to the notion of real time. Rather it is a relative counter. Every time the scheduler is entered it will increase by some constant unit. The variable *cupst* saves the elapsed time since the current active task *ct* was scheduled. Services for requesting real time can be provided later by the operating system and an external device.

[4] Although inter *task* communication would be the correct term here, for convenience we will go on using the term IPC.

5.2.2 The tasks

Tasks are similar to user processes. In addition to a local state they maintain information about their scheduling priority, privileges and a list of tasks willing to send messages to them. As processes, tasks can invoke services of the kernel via the trap instruction.

t_task = (tid : *nat*, localstate: *t_process*, timeslice : *nat*, resttime : *nat*,
priority : *nat*, priveleges : *bool*, communication: *t_comm*)

t_comm = (senderlist : *t_list_nat*, ipcpartner : *nat*, ipcto : *nat*)

- **Task Identifier**

  Every task has an unchangeable and unique ID.

- **The local state**

  The local state of a task is a Virtual DLX Assembler machine, as defined in Section 4.

- **Timeslice and Resttime**

  The variable *timeslice* denotes the time that is given to the task when scheduled. The variable *resttime* denotes the time left for the computations of task until another task is scheduled. Both variables can not be accessed by the task.

- **Priority**

  Every task has a scheduling priority. The detailed scheduling options are not explained here (for details see [10]).

- **IPC**

  Some data structures of interprocess communication are modeled as part of each task. In the variable *senderlist* all tasks are maintained that want to send a message to the task. All tasks in the *sendlist* are blocked and not scheduled until their request is served (they are sleeping). In the field *ipcto* the timeout value for the interprocess communication is specified.

- **Privileges**

  We distinguish two classes of tasks: privileged and non-privileged. Privileged tasks are also called system tasks (later, one of these system tasks will be instantiated to the operating system). System tasks are allowed to invoke all kernel calls. In contrast unprivileged tasks are only allowed to call the services *ipcSendReceive* and *getScheduleparam*.

5.2.3 The devices

Devices are modeled as finite-state machines.

t_vdevice =
(did: *nat*, *dspace*, *initspace* $\subseteq$ *dspace*, localstate : *t_dspace*, ports : *t_device*,
transition : *t_vdevice* $\times$ *t_vamosinput* $\longmapsto$ *t_vdevice* $\times$ *t_list_exint*)

A device is given through a unique ID, its state space, a space of valid start states, its current state, the state of the ports and a transition relation (a more elegant solution

would be a function taking some type and returning the transition relation, but this would need some more formal effort). The communication interface between devices and the kernel is still represented as in CVM* in the component *ports*. As finite state machines, devices can perform computations, according to some function *transition*. This function takes as input the state of the device and an input signal (representing unpredictable events as key pressed, etc.) and returns a modified device state and a list of occurred external interrupts.

5.3 Semantics

As in the CVM* model, we will again define the transition relation in VAMOS* as a composition of the transition relations of the modeled machines. The transitions of the kernel are now only represented by the *atomic* execution of the kernel calls and interrupt handlers.

5.3.1 The transition relation

- **Local Transitions of tasks**

  $\delta_{task} : t\_task \longmapsto t\_task$

  As in the case of processes, tasks perform local computations by executing the DLX assembler instruction at position *task.localstate.dpc*. δ_{tasks} manipulates *only* the local state of the task (in fact it is δ_{dlx} applied to the local state of the task).

- **Execution of kernel calls**

  The function *isService?* checks whether the next instruction of the currently active process is a trap instruction:

  $isService? : t\_vamosstate \longmapsto bool$

  If this is the case the service can be executed with the help of the function:

  $\delta_{service} : t\_vamosstate \longmapsto t\_vamosstate$

  Kernel calls can manipulate the global state of the model. In contrast to their C0 implementation, they are executed atomically. $\delta_{service}$ checks whether the called function and the number of transmitted arguments match with a kernel call.

- **Interrupt handling**

  The function *isInterrupt* indicates that either a local interrupt (other than trap) or an external interrupt occurred during the last transition (saved in the global variable *ex*):

  $isInterrupt : t\_vamosstate \longmapsto bool$

  All interrupts are handled by:

  $\delta_{interrupt} : t\_vamosstate \longmapsto t\_vamosstate$

  If an internal interrupt has occurred, $\delta_{interrupt}$ starts the local interrupt handler. In the case of an external interrupt (which is indicated in the field *ex*) the kernel performs some computations and sends an (faked) IPC message to the system task, coding all information of the respective interrupt. So the system task is expected to handle the interrupt. Given more than one interrupt one handler is

chosen deterministically.

$\delta_{interrupt}$ also contains the specification of the scheduler, as part of the semantics of the external interrupt handlers.

- **Transition of device**

  Every device is given through a state space and a transition relation. The transition relation can manipulate the local state of the device, the ports and the list of occurred external interrupts *ex*. The function δ_{device} executes one step of the device with the ID given as argument. It further takes as arguments the input signal (some unpredictable event) and the global state of the model and returns a state in which the executed device and the *ex* field changed.

$$\delta_{device} : nat \times t\_input \times t\_vamosstate \longmapsto t\_vamosstate$$

With the help of the presented functions, we can now define the transition relation δ_{vamos} of our model. If an interrupt occurred it will be handled by $\delta_{interrupt}$. $\delta_{interrupt}$ will handle external interrupts before handling internal ones. If no interrupt occurred the next instruction of the currently active task is executed. If this instruction is not a trap, the task performs a local computation. In the case of a trap instruction a kernel call is invoked and executed through $\delta_{service}$ in one step. Devices can always perform a transition. Through this non-determinism is introduced into our model.

Formally δ_{vamos} is given through:

Writing $\{\delta_{device}(k,i,c)\}$ denotes a set of states generated for all possible device IDs k.

$$\delta_{vamos} : t\_vamosinput \times t\_vamosstate \longmapsto 2^{t\_vamosstate}$$

$$\delta_{vamos}(i,c) = \{\delta_{device}(k,i,c)\} \cup \begin{cases} \{\delta_{service}(c)\} & : \text{ if } isService?(c) \wedge \overline{isInterrupt(c)} \\ \{\delta_{task}(c)\} & : \text{ if } \overline{isService?(c)} \wedge \overline{isInterrupt(c)} \\ \{\delta_{interrupt}(c)\} & : \text{ if } isInterrupt(c) \end{cases}$$

The conditions for the first three cases of transition (formulated in the right bracket) don't intersect, in contrast to the device transition, which is always enabled. By this nondeterminism is introduced into our model.

As mentioned in Section 2, internal transitions $\xrightarrow{\tau}$ can be annotated. We will use the annotations: $service_{kcd}^{tid}$ if task with ID *tid* invokes kernel call with number *kcd* and $local^{tid}$ if the transition was a local execution of a task with ID *tid*.

5.3.2 Example for kernel calls

In the following we will illustrate the kernel call mechanism by explaining *interprocess communication* in detail.

The communication is realized over the kernel calls *ipcSend*, *ipcReceive*, *ipcOpenReceive* and *ipcSendReceive*. A user task invokes one of these kernel calls by the assembler instruction *trap* (the function *isService?* becomes true). The parameters of the invocation are given at a predefined place in the *gpr* of the user process. Then, according to the immediate constant of the trap instruction the kernel handler will choose one of the four mentioned services and execute it (through $\delta_{service}$). Each kernel call checks

whether certain conditions are satisfied. If not an error message is reported. The calling task will sleep until a result is returned.

- **ipcSend**
  The parameters of the call are the ID of the receiver task, the length and start address of the message to transmit and a timeout value. The conditions are that the caller is a privileged task and that the receiver task exists. If the receiver task is in the sleep list, it has either invoked an *ipcOpenReceive* call or an *ipcReceive* from the sender task, the specified length of messages of both receiver and sender are identical and the message lies within the virtual memory of the sender, then the message is copied from the specified region in the memory of the sender task to the specified one of the receiver task. Furthermore sender and receiver tasks are informed about the successful transmission.

  If the receiver task has not invoked one of the mentioned receive calls the ID of the sender task is appended to the sender list of the receiver task and the sender task is set asleep.

  The timeout value is checked by the scheduler. If it expires the sending task is woken up with an error message.

- **ipcReceive**
  The parameters of the call are the ID of the sender task, the length and start address of the buffer to receive the message and a timeout value. The conditions are that the caller is a privileged task and that the sender task exists. If the ID of the sender task is in the sender list then the message is copied as described above (under the same conditions). Else the task is set asleep until it receives the desired message or the timeout expires.

- **ipcOpenReceive**
  Similar to *ipcReceive*, with the difference that the message is received from the first task in the send list.

- **ipcSendReceive**
  This is the only IPC kernel call that can be invoked by user tasks and hence is their only way of communicating with the rest of the system. The parameters of the call are the ID of the communication partner, the length and start address of the message to transmit, the length and start address of the receive buffer and two timeout values. If the caller is a unprivileged task the timeout values have to be infinite. The rest of the semantic is in fact given through a linking of the *ipcSend* and *ipcReceive* semantics.

The design of VAMOS IPC follows the assumption, that the system task never trusts an user task and that an user task always has to trust the system task. This leads to the conditions of infinite timeout values for *ipcSendReceive* on user task side.

Other kernel calls are not explained any further but a list of them is given in table 1.

Task Management	
*taskCreate	creates a new task
*taskClone	clones a given task
*taskKill	kills a task
*taskGetPrivileges	returns the status of a process
taskGetMyPid	returns the PID of the caller
*taskSwithTo	switches currently executed task
*taskChangeSchedulingParam	changes scheduling priorities
Memory management	
*memoryAdd	extends the virtual memory of a task
*memoryFree	shortens the virtual memory of a task
I/O-Devices	
*ioIn	copies data from tasks to device ports
*ioOut	copies data from device ports to tasks
Interprocess Communication	
*ipcSend	sends ipc-message
ipcSendReceive	sends ipc-message and waits for answer
*ipcReceive	receives ipc message
*ipcOpenReceive	receives ipc message

Table 1: Overview kernel calls — *indicates that the kernel call can only be invoked by the system task

5.4 Correctness

For VAMOS* we identified five main criteria of correctness for this layer: equivalence to the underlying model CVM*+VAMOS, integrity of tasks, privacy of tasks, fairness of scheduling and correctness of services. In the following we will present the formalization of three criteria and outline the other two. Proofs are only sketched.

Unfortunately we can not formulate equivalence, etc. directly over the model VAMOS* because it models *more* than CVM*+VAMOS, namely the devices. We will therefore refer with VAMOS* to the model without modeling external devices. Input signals in this variant of VAMOS* are the same as in CVM*.

5.4.1 Equivalence

In the first step we state the relation between the new model VAMOS* and the last model of the so far established sequence of models: CVM*+VAMOS. This is done by the equivalence theorem (see Section 2):

Theorem 2 The models CVM*+VAMOS and VAMOS* are equivalent under the relation VSIM.

Before proving equivalence we have to find the state relation VSIM between the two models. In the considered case, VSIM is mainly a mapping from C0 data structures of the implementation to components of *kds* in the state space of VAMOS*. Then the proof splits into two cases:

- **CVM*+VAMOS implements VAMOS* under VSIM** This could be proven with structural induction over all types of transitions. We have to show, whenever a transition could be taken in the VAMOS* model this transition could be simulated by arbitrary many steps in CVM*+VAMOS.

  In the case of a local transition of a task nothing has to be shown since nothing changed in respect to CVM*+VAMOS.

  When the transition in VAMOS* is the execution of a kernel call, we have to show two things: that kernel calls in CVM*+VAMOS can not be interrupted (are completed before another process is scheduled) and that the code segment related to the kernel call really implements the specification. In the first part a.o. one has to prove that no local interrupt occurs in the kernel code. The second part is mainly C0 code verification.

  The last relevant transition type is given by interrupt handling through the kernel. As for kernel calls we have to apply C0 code verification.

- **VAMOS* implements CVM*+VAMOS under VSIM^{-1}** This sub-proof mainly depends on the fact, that no two different interrupts of the same type may occur during the execution of a kernel call or kernel handler. By this we know that interrupts may not occur faster in CVM*+VAMOS than in VAMOS*.

The equivalence proof gives us some useful properties as by-products. One of these by-products is the *availability* of the kernel services. The currently active task can just by definition of VAMOS* assume that the kernel will provide the invoked service.

Obviously the equivalence theorem is not enough. We rather have to prove that our specification *makes sense*. All following properties can be formalized and proven in the VAMOS* model without resorting to CVM*+VAMOS. Note that there is no automatic method for finding all necessary properties, therefore this list may be not complete. Probably we will need to add further properties as they are requiered by higher level models.

5.4.2 Integrity of tasks

Intuitively this theorem states that no hacker can manipulate the state of a task. I.e. during a run the state of a task is either altered after a computation of this task, or after the invocation of a kernel call. This kernel call must either be an IPC message or a memory add, a memory free or kill request with a privileged task as caller.

Theorem 3 Integrity of tasks For all runs (σ, α) of the VAMOS* model and for all reached states $\sigma[i]$ of the run the following holds:

The constants $const\_KCmemAdd$, $const\_KCmemFree$, $const\_KCkill$, $const\_KCIPCsend$ and $const\_KCIPCreceive$ represent the numbers for the kernel calls *memoryAdd*, *memoryFree*, *taskKill*, *ipcSend* and *ipcReceive*.

$\forall t \in \sigma[i].tasks$ with tid $=$ *t.tid*: $\sigma[i].tasks[tid] \neq \sigma[i+1].tasks[tid] \Longrightarrow$

$$\alpha[i] = local^{tid} \vee$$

$$\alpha[i] = service_{kcn}^{tid} \text{ for some } kcn \vee$$

$$\begin{aligned}(\alpha[i] = service_{kcn}^{cid} \ \wedge ((&\sigma[i].tasks[cid].priveleges = true \wedge \\ &(kcn = const\_KCkill \\ &\vee kcn = const\_KCmemAdd \\ &\vee kcn = const\_KCmemFree)) \vee \\ &((kcn = const\_KCIPCsend \vee kcn = const\_KCIPCreceive) \\ &\quad \wedge \sigma[i].tasks[tid].localstate = \sigma[i+1].tasks[tid].localstate)))\end{aligned}$$

The proof should be straightforward. Mainly one has to show that in all not mentioned cases kernel calls won't manipulate the local state of a task.

5.4.3 Privacy of tasks

So far we only stated that no hacker can manipulate another task. For proving the security of our system we need to show that no hacker can *read* any data of the task. Obviously we have to assume that the malicious task is not the system task, or that the system task is restricted to invoke only IPC kernel calls (and not for example *memFree*). In the following we will use the second assumption.

We will start with a very weak form of privacy of an *isolated memory cell* of a task. A memory cell is *isolated* when the owning task never reads the content of the cell. The first theorem says that the content of such an isolated task can not be inferred by any hacker (including the owning task).

Definition Isolated memory cell Given a run (σ, α) of VAMOS*. For a task with ID *tid* of $\sigma[0].tasks$ the memory cell with address *ad* is called isolated iff:

The constants $const\_ADstart$ and $const\_ADend$ stand for the start and end addresses of the registers containing the arguments of the invoked kernel call.

$\forall i \in nat$:

$$\alpha[i] = service^{tid}_{kcn} \implies ad \notin \{\sigma[i].tasks[tid].localstate.gpr[const\_ADstart], ..., \sigma[i].tasks[tid].localstate.gpr[const\_ADend]\}$$

$$\wedge\alpha[i] = local^{tid} \implies head(\sigma[i].tasks[tid].localstate.program) \text{ does not refer to the memory cell with address } ad$$

With this, obviously too restrictive definition of isolated memory cells, we can have a first try, defining what privacy is.

Theorem 4 Weakest Privacy Given a run (σ, α) of VAMOS* and a task with ID *tid* of $\sigma[0].tasks$ with an isolated memory cell with address *ad*.

For all runs (σ', α) of VAMOS*, for which $\sigma'[0]$ differs from $\sigma[0]$ only in the content of the memory cell with address *ad* of task with ID *tid* the following holds:

$\forall i \in nat$: $\sigma[i]$ differs from $\sigma'[i]$ only in the local state of the task with ID *tid*

The proof depends mainly on the isolation of virtual memory spaces of processes of CVM* and on the correctness of implementation of the kernel calls. Indeed this privacy property is very weak. It states that no malicious task is able to infer any knowledge of some isolated memory cell of another task *if this task is not even allowed to read the memory cell*. The necessity of this strong condition lies in the possibility of tasks to infer information through the system time. Theoretically one or more cooperating tasks could infer if a another process is computing or not only by knowing the time passed between two invocations.[5]

One could introduce scheduling with constant slots for each task. This would lead to predictable execution times, without any information flow when requesting time.

We will take another approach by denying user tasks from both specifying a timeout value other than zero or infinite for their calls and from requesting the kernel time (this condition is directly given for all unprivileged tasks). With this restriction we could formulate privacy about the whole local state of a task, if this task does not call any services:

Definition Isolated task Given a run (σ, α) of VAMOS*. A task with ID *tid* of $\sigma[0].tasks$ is called isolated iff $\neg\exists i : nat.\ \alpha[i] = service^{tid}_{kcd}$.

Next we need a notion of runs *purged* of local actions of the isolated task. The purging function takes a run σ and deletes all states in the state sequence of σ which where reached directly after a computation of the isolated task.

[5] Indeed they must have knowledge of the detailed architecture. Furthermore it is not clear how many tasks must cooperate to infer any data of some other task

Definition Purging a run The recursive purging function deletes state changes caused by the isolated task.

fun purge $(nil, nil, tid) =$ nil
fun purge $(\sigma, \alpha, tid) =$ IF $\alpha[0] = local^{tid}_{kcd}$ THEN
purge$(tail(\sigma), tail(\alpha), tid)$
ELSE
head(σ) @ purge$(tail(\sigma), tail(\alpha), tid)$

With that we can formulate a stronger notion of privacy:

Theorem 5 Weak Privacy Given a run (σ, α) of VAMOS* and an isolated task with ID *tid* of $\sigma[0].tasks$. Further we assume that during the run all tasks, neither invoke a kernel call with timeout nor request the kernel time. Then the following holds:

For all runs (σ', α') of VAMOS*, for which $\sigma'[0]$ differs from $\sigma[0]$ only by local state of task with ID *tid* and where α' and α only differ in the annotation of local transitions the following holds:

$\forall i : nat.\ purge(\sigma, \alpha, tid)[i]$ differs from $purge(\sigma', \alpha', tid)[i]$ only in the local state of the task with ID *tid*.

The described way of defining *information flow* is based on noninterference models (for details see [11]). An object a is called non interfering with an object b if the actions of object a have no influence on the succeeding actions of object b.

Theorem 5 holds because the order of execution of the tasks is independent of the amount of computation caused by one isolated task. It not only says that no hacker can infer any information but also that even if the isolated task would be cooperating with a group of tasks, no information will propagate from his local state outside.

So far we had three conditions for privacy: privileged user tasks execute no services except IPC, tasks can not access time information (this condition is always fulfilled for unprivileged tasks) and the considered private task performs only local transitions. In our last privacy theorem we will relax the third condition by allowing services, except of communication with devices. For this we have to define a group of tasks only communicating with themselves.

Definition Isolated Group Given a run (σ, α) of VAMOS* and a set G of tasks in $\sigma[0]$. The set G is called isolated if no task in G sends an IPC message to a task outside G.

Next we have to extend the purging function to sets of tasks.

Definition Group Purging The group purging function does not consider state changes caused by tasks of an isolated group.

fun gpurge $(nil, nil, group) =$ nil
fun gpurge $(\sigma, \alpha, group) =$ IF $\alpha[0] = local^{tid}_{kcd}$ OR $\alpha[0] = service^{tid}_{kcd}$ AND tid $\in$ g
gpurge$(tail(\sigma), tail(\alpha), tid, i)$
ELSE
head(σ) @ gpurge$(tail(\sigma), tail(\alpha), tid, i)$

With that we can formulate a strong privacy property of tasks. Theorem 6 says that

no information flows into or out of an isolated group.

Theorem 6 Strong Privacy Given a run (σ, α) of VAMOS* and an isolated group G of tasks in $\sigma[0]$. For all runs (σ', α') of VAMOS*, for which $\sigma'[0]$ differs from $\sigma[0]$ only by the states of tasks in G and where α' and α only differ in the annotation of local transitions the following holds:

$\forall i : nat.\ gpurge(\sigma, \alpha, g)[i]$ differs from $gpurge(\sigma', \alpha', g)[i]$ only in the local states of tasks in G.

We still have two main restrictions, namely the restrictions on the system task and on requesting time. The first restriction can only be relaxed when certain conditions regarding the system task hold. Indeed we expect that we could design an operating system on top of our stack for which the condition holds, so that we can formulate a *fundamental theorem of operating system security* in our proceeding works.

We are aware that much more study of terms as *privacy* of applications, and *information flow* in our framework is necessary. The stated theorems should only give some hints on what is probably possible and what not.

5.4.4 Correctness of kernel calls (IPC)

The equivalence theorem only showed data structure correspondence between implementation and specification of kernel calls. It does not show that the specification itself makes sense. For this we have to examine every kernel call. In the following we will give an example of important properties for the IPC mechanism of VAMOS*.

Definition VAMOS IPC Correctness VAMOS IPC is correct if the following holds:

- If a rendezvous happens the message that was sent is received without change.
- Only messages that were sent are received.
- The order of messages is retained, i.e. messages are received in the order they were sent unless the receiver opts to receive from specific senders.
- Processes waiting for a rendezvous are blocked and never woken up unless the rendezvous can be completed or a specified timeout occurs. In case of a time out all traces of the IPC call are removed.

5.4.5 Fairness of Scheduler

Fairness of scheduler is a crucial property. In the succeeding models, only fair runs will be considered. There are many definitions of fairness we have chosen the following (not formally stated one):

Definition Scheduler Fairness All tasks with the same priority get equally often scheduled and receive the same amount of computation time unless:

- they terminate early,
- are blocked through IPC, or
- donate their time to other tasks.

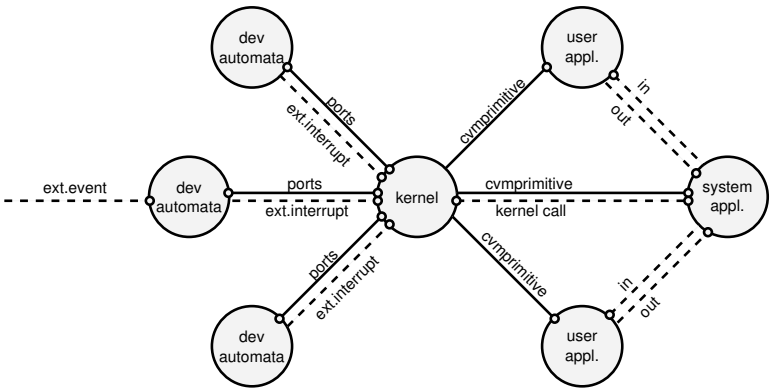

Figure 8: Communication Structure of VAMOS*+C0

6 Communicating Applications

The parameters in VAMOS* are the system task and the unprivileged tasks. The next step would be to "plug into" VAMOS* an operating system for the system task. The operating system is expected to provide all necessary services like communication with other tasks, access to hardware and rights management. Our operating system is the Simple Operating System (SOS). It is written in C0. It stands to reason, that also the correctness proofs of this operating system are related to C0-code and not to translated assembler instructions. The same is true for proving correctness of tasks. Therefore we first have to extend our model with C0-machines, such that tasks can either be C0-machines or Virtual DLX Assembler machines.

6.1 An intermediate step: VAMOS*+C0-Account

Extending our model with C0-machines [6] is not trivial, because C0 semantics has to be used in a concurrent system. We want to execute C0-statements as atomic units, although they are in fact translated into many different assembler instructions, which are by no means executed atomically.

6.1.1 The model

At first we will introduce the new model VAMOS+C0Account. With the help of this model we could show, that the atomic execution of a block of DLX-Assembler instructions (representing one C0-statement) changes the global state of the system as the stepwise and concurrent execution of the separate instructions.

Model definition: The account model

$$VAMOS^*{+}C0Acc = (t\_vc0astate, t\_vc0ainput, t\_vc0astart, \delta_{vc0a})$$

and the state definition:

t_vc0astate = (kds : t_kds, etasks : t_etasks, ..)

[6] In fact this step is both an abstraction and an instantiation. Functional abstraction because we use abstract C0 semantics and instantiation because we consider programs generated by a specific compiler.

With *kds, ct, devices* and *ex* defined as in VAMOS*. In VAMOS*+C0Acc the local state of a task is either given through the state a Virtual DLX Assembler machine or through an extended C0-machines. These extended C0-machines has in addition an *account* variable.

t_etask = (localstate: t_process ∪ t_ec0machine, ...)

t_ec0machine = (c0m : t_c0machine, account : nat)

6.1.2 Semantics

The transition relation of VAMOS*+C0Acc differs from the transition relation of VAMOS* only in the transition of the extended C0-tasks. We associate with every C0-statement in a program a *cost*, i.e. the count of DLX Assembler instructions that are executed for the translated C0-Statement. If a C0 task is scheduled and it has on its *account* accumulated more or equal instruction time as the cost for the next C0-statement, the statement will be executed. Else the *account* is increased by one.

$\delta_{etask} : t\_etask \longmapsto t\_etask$

The expression $[x := c]$ in the left side of the defining equation means, that the result configuration is the given one where the component x is substituted by the value c.

$definels = t.localstate$

$$\delta_{etask}(t) = \begin{cases} \delta_{task}(t) & : \text{if t is DLX machine} \\ \delta_{c0}(ls.c0m) & : \text{if t is C0-machine} \wedge \\ & \quad \text{t.account} = \text{cost(t.c0m.prog[0], t)} \\ [t.account := t.account + 1] & : \text{if t is C0-machine} \wedge \\ & \quad \text{t.account} < \text{cost(t.c0m.prog[0], t)} \end{cases}$$

Furthermore the semantics of the kernel calls (see $\delta_{services}$ 5.3) have to be adapted to C0-programs. For calling a service we extend the language C0 with so called *communication primitives*. These primitives are translated by the compiler to the associated trap instruction. These changes are all straightforward.

6.1.3 Correctness

The *cost* function can not always be computed statically, it rather could depend on the concrete state of the C0 machine. Indeed it coincide with the construct *s(t)* in the proof of the C0 compiler correctness theorem (for details see [17] p. 11): *t* steps of the C0 machine are simulated by *s(t)* steps of the DLX machine. With that our *cost* for the C0 statement executed at position *t* in the C0 program is given through *s(t+1) - s(t)*.

The relation between the account model and VAMOS* is given through equivalence:

Theorem 7 Equivalence VAMOS*+C0Acc and VAMOS* are equivalent under the relation C0ASIM. C0ASIM maps each C0-machine configuration of the abstract model to the DLX Assembler machine configuration generated by the C0 compiler; for other components C0ASIM is the identity.

This theorem states compiler correctness extended to concurrent systems. The proof mainly depends on the *integrity* theorem (see 5.4.2) of user tasks. By this theorem

we know that an interrupted execution of a C0 statement (by the scheduler) will be resumed without a manipulation of the local state.

6.2 The model VAMOS*+C0

Obviously we are interested in a model with *pure* C0-machines, *without* an account variable, especially when programs are verified. But by ignoring the account we are at the same time losing (nearly) all information about the order of execution. We get a *temporal* abstraction without a scheduler. Instead all possible and fair executions will be considered.

This is a major step because we change the semantic interpretation of our model. Indeed we will see that we only can show the completeness of our new model, but not the correctness. Is this a problem? We want to state assertions over the behavior of programs in an operating system environment. In most cases the correctness of functionality and security properties does *not* depend on one *particular* scheduler. Only certain assumptions about the scheduler are needed. That means, one expects that a program behaves correct under all possible and fair schedulers.

For some applications, needed properties can only be shown for certain orders of executions as in the automotive control system OSEKtime [23]. For such purposes the model VAMOS*+C0Acc can be used and extended to a new development line.

So far we have seen in the model VAMOS*+C0Acc the

- **Functional Abstraction:** Introducing C0-semantics

and sketched the resulting

- **Temporal Abstraction:** No more scheduling is considered

The third abstraction is structural and linked to interprocess communication (IPC). In VAMOS*+C0 IPC is modeled a bit different than in VAMOS*: IPC is represented as shared variables (this concept is a simplification of [27]). By this all IPC functionality (and all necessary data structures) are modeled as part of the tasks. These extended tasks will be called *applications* in the following.

Formally the model is given through:

Model definition: Communicating applications

$$VAMOS*+C0 = (t\_vamosc0state, t\_vamosc0input, t\_vamosc0start, \delta_{vamosc0})$$

6.3 State Space

A state of the model consists of the set of states of all user applications and the state of the system application. User applications are related to the unprivileged tasks and the system application is related to the system task in VAMOS*. Devices are modeled exactly as in VAMOS*.

t_vamosc0state = (userapps : t_list_userapp, sysapp : t_sysapp,
devices : t_list_vdevice, ex : t_list_exint)

6.3.1 User application

User applications consists of a local state, a communication part, a status and an alive bit. The local state is either given through a Virtual DLX Assembler machine or through a C0-machine. The communication part represents the interprocess communication functionality of the application. *alive* indicates whether the application is alive or not, i.e. whether it can be scheduled or not.

t_userapp = (localstate: t_process ∪ t_c0machine, comm: t_comm, status: bool, alive: bool)

The communication structure consists of an input and an output variable shared with the system application. The type of both variables is the IPC message type *t_complex* (in 7.3 the type is described in detail). The sending task writes its IPC message into its *out* variable, and waits at the *in* variable for answers. *Status* indicated whether the communication finished (explained in 6.4.1). *Order* is only relevant for the system application.

t_comm = (shared: t_shared, status: bool)
t_shared = (in : t_complex, out : t_complex, , order: nat)

C0 user applications can manipulate their communication part by the special primitive *callService(sendbuf, recvbuf)*, which can be a statement in the program rest (for details see 6.4.1).

6.3.2 System Application

The system application is a C0-machine. The system application is allowed to call all kernel calls and in particular to send and receive messages from every other application. Therefore it has not only one pair of input and output variables, but an array of pairs. For each application with ID x it maintains exactly one pair at position x of the array. The pair at position zero is dedicated for handling external interrupts by the system application (only the input variable is used).

From Section 5.4.5 we hope to show that IPC is implemented in a fair manner. That means a message sent to a process performing infinitely often *ipcOpenReceive* (as we will see later for the operating system) will finally be delivered. To preserve this property we must remember the order of incoming send requests. This is realized by the variable *order*. It saves the order with which new messages arrive at the system application.

t_sysapp = (localstate: t_c0machine, comm: t_commsys)
t_commsys = (shared: t_list_shared, status: bool)

For manipulating the communication part the user application can use the primitives *receiveService(res, recvbuf, to)*, *sendTo(res, sendbuf, sid, to)*, *receiveFrom(res, recvbuf, rid, to)* and *sendReceive(res, sendbuf, recvbuf, sid, to)*. For a detailed explanation see 6.4.1. These primitives are used as ordinary statements in the C0 program and are translated by the compiler to the corresponding DLX Assembler instruction (trap

instruction, etc.).

One type is introduced for both user applications and the system application.

t_application = t_userapp ∪ t_sysapp

6.4 Semantics

The transition relation of the new model is fairly similar to that of VAMOS*+C0Acc. For the new abstraction of IPC we have to define a notion of consistent states, which captures the shared variable semantics of the new model.

Definition of IPC-consistency A state *c* is called IPC-consistent, iff for each user application with ID x $c.sysapp.comm.shared[x] = c.app[x].comm.shared$ holds.

The semantics of all our functions must be defined in such a way that IPC-consistency is always preserved. Furthermore the transition relation $\delta_{vamosc0}$ is restricted to *fair* executions only.

6.4.1 Transitions

- **Local transitions of applications**

  An application can perform local transitions when the next instruction/C0 statement is not a trap instruction/communication primitive. Local transitions of applications are either transitions of a Virtual DLX Assembler machine or of a C0 machine.

  $\delta_{localapp} : t\_application \longmapsto t\_application$

- **Communications of applications**

  The function *isIPC?* checks whether the next instruction/C0 statement of the currently active application is a trap instruction/communication primitive:

  $\delta_{isIPC?} : t\_application \longmapsto bool$

  If this is the case the semantics of the communication primitive or trap instruction of the application is applied through the function δ_{comm}:

  $\delta_{comm} : t\_vamosc0state \times t\_application \longmapsto t\_vamosc0state$

  Because the kernel and all its data structures are not anymore modeled (especially the sleep list), the blocking conditions of IPC must be coded in some way in the shared variable concept. This is done through the new semantics of the communication primitives. Therefore the semantics of a communication primitive consists of a *condition* and an *action* (see below 6.4.2).

- **Execution of kernel calls**

  The system application is allowed to invoke all kernel calls provided by VAMOS. The following function determines if the next statement in the C0 program of the system application is an invocation of VAMOS:

  $\delta_{isService?} : t\_sysapp \longmapsto bool$

  If this is the case the service is executed with the function:

$\delta_{service} : t\_vamosc0state \longmapsto t\_vamosc0state$

$\delta_{service}$ is the same function as in VAMOS* adapted to the data structures of the new model.

- **Interrupt handling and transition of devices**

  Interrupts, devices and their transitions are treated as in VAMOS*. External interrupts are handled by sending an IPC message to the system application (writing it into the dedicated input variable of the system application).

6.4.2 δ_{comm} in detail

δ_{comm} consists of the semantics of the communication primitives and the trap instructions. In the following we will give the detailed definition of the two most important communication primitives for C0 programs. When introducing the operating system we will use the primitive *callService* for calling services of the operating system from a user application. *receiveService* is used by the operating system to receive all incoming calls for services. The definitions of the other primitives are similar.

- The primitive *callService* is translated to an *ipcSendReceive* VAMOS call. When an user task calls *ipcSendReceive* the timeout value must be infinitely and the receiver must be the privileged task. Therefore our new primitive only takes two arguments: the message to send and the place where to copy the answer. Here we have to note that *no* VAMOS result is returned. This design is justified, because we can prove that no error case of this call can occur. The error cases mentioned in 5.3 occur if a timeout expire (timeout value is infinite), if the counterpart of communication is killed (we assume that the operating system is always alive) or if the receive or send buffers are not in the virtual memory of the caller (we only read and assign given C0 variables).

  The primitive blocks in the case the message is sent (i.e. written in the output variable), but no answer is received yet. *Action* consists of two parts: the sending of the message containing the system call and the receiving of the answer.

The notation *localstate.memory:x* := *c* means we assign the C0 variable *x* to value *c*.

$\delta_{callService}$**(c, sid)** $\stackrel{def}{=}$

CONDITION
head(c.userapps[sid].localstate.program) = callService(sendbuf, recvbuf) ∧
c.userapps[sid].comm.status ⟹ (c.userapps[sid].comm.shared.out = empty
∧ c.userapps[sid].comm.shared.in ≠ empty)

ACTION
¬(c.userapps[sid].comm.status) ⟹ //message not sent yet
(c.userapps[sid].comm.shared.out := sendbuf ∧
c.sysapp.comm.shared[sid].in := sendbuf ∧
c.userapps[sid].comm.status := true ∧
c.sysapp.comm.shared[sid].order := $\underbrace{max}_{i}${c.sysapp.comm.shared[i].order} +1
∧ c.userapp[sid].comm.shared.order := c.sysapp.comm.shared[sid].order)

c.userapps[sid].comm.status ⟹ //message was sent and in receive state
(c.userapps[sid].localstate.memory:recvbuf := c.userapps[sid].comm.shared.out

∧ c.userapps[sid].comm.shared.in := empty
∧ c.sysapp.comm.shared[sid].out := empty
∧ c.userapps[sid].comm.status := false ∧
c.userapps[sid].localstate.program := tail(c.userapps[sid].localstate.program))

- The primitive *receiveService* is translated to an *ipcOpenReceive* VAMOS call. In contrast to the user applications the system application can specify for communication primitives a timeout value *to*. Therefore also VAMOS errors can occur during executing the primitives. The VAMOS result is saved in the variable *res*. Because we do not model kernel time at all, the timeout expires non deterministically in our new model.

The notation *localstate.memory:x := c* means we assign the C0 variable *x* to value *c*. The constant *const_IPCTIMEOUT* indicates that the specified timeout expired. The constant *const_IPCRECVSUCC* indicates that a message was successfully received.

$\delta_{receiveService}$**(c)** $\stackrel{def}{=}$

CONDITION
head(c.sysapp.localstate.program) = receiveService(res, recvbuf, to)

ACTION
define sid = $\underbrace{argmax}_{i}${c.sysapp.comm.shared[i].order}

c.userapps[sid].comm.order = 0 ⟹ // *nothing to receive - modeled as timeout*
(c.sysapp.localstate.memory:res := const_IPCTIMEOUT ∧
c.sysapp.localstate.program := tail(c.sysapp.localstate.program))

c.userapps[sid].comm.order ≠ 0 ⟹
(c.sysapp.localstate.memory:recvbuf := c.sysapp.comm[sid].shared.in
c.sysapp.localstate.memory:res := const_IPCRECVSUCC
∧ c.sysapp.comm.shared[sid].in = empty
∧ c.userapps[sid].comm.out := empty
∧ c.sysapp.comm.shared[sid].order := 0
∧ c.userapp[sid].comm.shared.order := 0
∧ c.sysapp.localstate.program := tail(c.sysapp.localstate.program))

6.5 Transition relation of the system

With the help of all above declared functions we obtain the transition *relation* for the Model VAMOS*+C0. An application can perform either a local computation or a communication transition. Same holds for the system application. Furthermore the system application can invoke kernel calls other than IPC related ones.

Formally the transition relation is given through:

The expression $[x := c]$ in the left side of the defining equation means, that the result configuration is the given one where the component x is substituted by the value c. The function *getId* returns the ID of an application. Writing $\{\delta_{userapp}$*(c, userapps[k])}, if cond* denotes a set of states generated for arguments satisfying the condition *cond*.

$$\delta_{userapp} : t\_vamosc0state \times t\_userapp \longmapsto t\_vamosc0state$$

$$\delta_{userapp}(c, app) = \begin{cases} [c.userapps[getId(app)] := \delta_{local}(app)] & : \text{ if } \overline{isIPC?(app)} \\ \delta_{comm}(c, app) & : \text{ if } isIPC?(app) \end{cases}$$

$\delta_{sysapp} : t\_vamosc0state \longmapsto t\_vamosc0state$

$$\delta_{sysapp}(c) = \begin{cases} [c.sysapp := \delta_{local}(c.sysapp)] & : \text{ if } \overline{isIPC?(c.sysapp)} \\ & : \wedge \overline{isService?(c.sysapp)} \\ \delta_{comm}(c, c.sysapp) & : \text{ if } isIPC?(c.sysapp) \\ \delta_{service}(c) & : \text{ if } isService?(c.sysapp) \end{cases}$$

$\delta_{vamoc0} : t\_vamosc0state \times t\_vamosc0input \longmapsto 2^{t\_vamosc0state}$

$$\delta_{vamosc0}(c, i) = \{\delta_{device}(c, i)\} \cup \begin{cases} \{\delta_{userapp}(c, c.userapp[k]), \\ \quad \delta_{sysapp}(c)\} & : \text{ if } \overline{isInterrupt(c)} \wedge \\ & c.userapp[k].alive \\ \{\delta_{interrupt}(c)\} & : \text{ if } isInterrupt(c) \end{cases}$$

Obviously $\delta_{vamosc0}$ is non deterministic. C0-Statements are treated as atomic units. In contrast to the kernel the system application can be interrupted and computations of other applications can interleave with its execution.

6.6 Correctness of VAMOS*+C0

Obviously we can not hope to prove equivalence between VAMOS* and VAMOS*+C0. That is because VAMOS*+C0 contains *all* fair runs, not only those produced by the VAMOS scheduler. We only claim *completeness* of the new model, i.e. that all runs of VAMOS*+C0Acc can be simulated by VAMOS*+C0 under a proper state relation C0SIM.

Theorem 8 Completeness of VAMOS*+C0 VAMOS*+C0 implements VAMOS*+C0Acc under C0SIM.

The simulation relation C0SIM is mainly a mapping between the abstract data structures for modeling IPC in both models. When designing a model for which only completeness holds, we must be extremely careful not to loose to much expressiveness. For example modeling IPC as shared variables without order would lead to a loss of the fairness property of IPC, although the completeness relation holds.

It is worth mentioning that completeness of VAMOS*+C0 could be directly related to VAMOS* without introducing VAMOS*+C0Acc. One would only need C0-compiler correctness and the non interfering property of the kernel. We took the intermediate step with the account model for two reasons: for a better understanding and for showing how one could design a model by C0 semantics and without losing the order of execution.

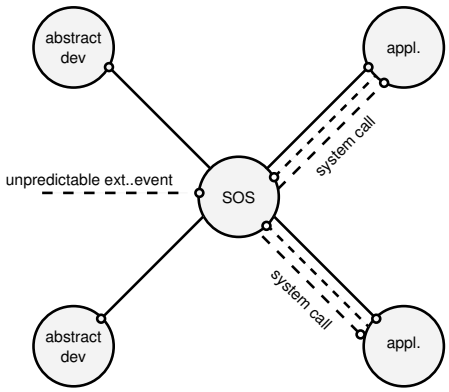

Figure 9: Communication Structure of SOS*

7 Communicating Applications In The OS

7.1 The model

After showing how C0 semantics is introduced in our framework, we now can "plug in" the *Simple Operating System* (the *sos-machine*, for implementation details see [6], for modeling see [5]) into VAMOS*+C0. We obtain the new instance VAMOS*+C0+SOS. The main code of SOS is given by the implementation of the so called SOS *system calls*. SOS *system calls* provide the user applications with all necessary services as communication and the file system.

System calls are invoked through VAMOS IPC (through the communication primitive *callService*) messages to the system application. The system application waits with the help of the primitive *receiveService* for new calls. Whenever the timeout of this primitive expires some internal data structures (as SOS timeouts) are updated and again *receiveService* is invoked.

Furthermore the handlers for external interrupts are implemented (the drivers). These handlers are as described in Section 5 also invoked through VAMOS IPC messages from the kernel.

Model definition: SOS Instantiation of Communicating Applications

$$VAMOS + C0 + SOS = VAMOS + C0[sysapp := sos - machine]$$

As for CVM*+VAMOS the concrete C0-code of the Simple Operating System is not of interest for higher layers. In the next step we will therefore introduce the new model SOS*. SOS* contains data, functional and structural abstraction:

- **Data abstraction** Only the user visible state of devices should be modeled. The hard disk for example is modeled as part of the file system.
- **Functional abstraction** The C0 code of the system calls is only represented through its effects on the SOS data structures and on the global state. Furthermore each system call is represented as one atomic step (although in the underlying model the execution may even be interleaved with the executions of other applications). This leads to structural abstraction.
- **Structural abstraction** Complex communication protocols between the operating system and devices (as read from hard disk) are modeled as one atomic step.

SOS* will be our top-level model. It can be instantiated by user applications, which one wants to verify. The model is given through:

Model definition: Communication Applications in SOS

$$SOS* = (t\_sos, t\_sosinput, t\_sosstart, \delta_{sos})$$

Because devices are no longer modeled as finite-state automata which can generate external interrupts, the input to our new model are unpredictable external interrupts.

7.2 The state space

The state space of SOS* is fairly similar to that of VAMOS*+C0. It consists of data structures representing the operating system, a set of user applications, a set of abstract devices and a field storing unpredictable external interrupts.

t_sosstate = (userapps : *t_list_userapp*, sosds : *t_sosds*, devices: *t_list_absdevice*, ex : *t_list_exint*)

User applications are modeled as in VAMOS*+C0. The operating system (or formerly the system application) is now only represented through some needed data structures, and is not anymore modeled as C0 machine.

7.2.1 Operating System data structures

The component representing the operating system consists of a local state and a communication state. The communication state is the same as the one of the system application in VAMOS*+C0.

The local state of the operating system maintains data structures needed for executing the system calls. Furthermore some data structures save information about devices and *users*. An user can own *resources*. With each device, process and user a resource is linked describing access permissions to the object and the ownership. User applications *request* to control resources. Granted requests are called *controls*.

With that we get for the definition of the SOS data structures:

t_sosds = (localstate: *t_soslocal*, communication: *t_list_comm*)

t_soslocal = (chardevs : *t_list_chardev*, users : *t_list_user*, resources : *t_list_resource*, requests: *t_list_controls*, controls: *t_list_control*, time: *nat*)

- **Character devices**

  In this data structure the operating system maintains information about all character devices as the monitor and files. For each character device the type (e.g. file, screen, etc.), the name (e.g. filename), size, the current offset, a bit indicating if the size is extend able and the set of valid chars are saved. This data structure is not to be confused with the abstract device. In the abstract device the whole state of the devices (e.g. the content of a file) is saved.

  t_chardev = (type : *nat*, name : *string*, size : *nat*, offset : *nat*, extendable : *bool*, chars : *t_set_char*)

Example for a file as character device where ASCII is the set of all ASCII characters:
(type = const_CT_FILE, name = "shell.exe", size = 42000, offset = 0, extendable = true, chars = ASCII)

- **User**

  Properties of a user are its ID, the password, and the filename of the file containing the program to start upon login.

  t_user = (uid: nat, password: string, loginshell: string)

  Example for an user
  (uid = 4, password = "verisoft", loginshell = "shell.exe")

- **Resources**

  With every application, user, file, screen, etc. a resource is linked.The resource specifies the type (e.g. file), the name (e.g. filename), the owner (the owner has to be a user) and the list of access rights, called *permissions*. Permissions consists of the type of the access right (e.g. file read) and a list of ID of users that possess the right.

  t_resource = (type: nat, name: string, owner: nat, access: t_list_permission)
  t_permission = (accesstype: nat, uids: t_list_nat)

  Example for a resource of a file, specifying only the permission for user with ID 4 to read the file "shell.exe":
  (type = const_RT_FILE, name = "shell.exe", owner = 4, access = [permission1])
  permission1 = (type = const_COP_READ, uids = [4])

- **Requests**

  User application request resources over system calls. Each request is saved as an item of type control in a request list. Control data structures over resources consist of a type, a name, the type of the control (e.g. read file), the application ID, and a deadline, a time when the request expires (relative to SOS time).

  t_control = (type: nat, name: string, controltype: nat, aid: nat, deadline: nat)

  Example of a request from the application with ID 3 for reading the file "shell.exe":
  (type = const_RT_FILE, name = "shell.exe", controltype = const_COP_READ, aid = 3, deadline = 100)

- **Controls**

  If a request is granted it is moved from the request list to the control list.

- **Time**
  The variable *time* increases every time the SOS is entered.

7.2.2 Abstract Devices

In SOS* devices should only be represented through the user application visible state. Transitions should not be visible and all functionality should be hidden completely behind designated system calls. For example the hard disk is only modeled as a mapping between filename and content and not as word addressed memory. The complex bus communication protocol is no longer visible. Monitor, keyboard and the file system are concrete abstract devices, all other devices are parameters of the model. From all external interrupts only those which are unpredictable are modeled. These unpredictable interrupts are stored in the field *ex* and are as in VAMOS*+C0 sent as a message to the system application. Then the system application chooses the corresponding handler.

7.3 Semantics

7.3.1 SOS System Calls

System calls provide the user application with all necessary services. For that system calls can manipulate the global state of the model. In SOS* they are treated as atomic steps. System calls are invoked by user applications over the communication primitive *callService*.

The messages for invoking system calls and for the return values of the call are encoded in the type *t_complex*. The number of the system call to invoke is specified in the field *syscall*, the ID of the caller application is stored in *source*. Arguments of the system call are encoded in the field *argnat*, *argint* and *argchar*. In the case the system call is a SOS IPC, the payload of the message is not saved in *arguments* but in the field *ipc*. If the message is a result of a system call, the field *result* indicates whether the call was successful or not.

t_complex = (argnat: t_array_nat, argint: t_array_int, argchar: t_array_char, result: nat, ipc: t_ipc, source: nat, syscall: nat, timeout: nat) $\cup$ {empty}

The message with the system call is written into one of the input variables of the operating system and the application blocks until the answer is sent (also in form of a *t_complex* message). The operating system will read the incoming message with the call after a finite count of rounds (because of the fairness assumption over runs and the *order* construct in the shared variables). If it is a well formed call (system call number is correct, arguments are well typed, etc.) the call is executed through $\delta_{syscall}$.

$\delta_{syscall} : t\_sosstate \longmapsto t\_sosstate$

The function $\delta_{syscall}$ serves three roles:

- **Check and preparation of data structures** The variable *time* is increased by one and the SOS data structures are checked for expired timeouts.
- **Execution of system calls** In that step local SOS data structures and then the global state are manipulated.
- **Execution of handler for external interrupts** If the received message is sent by the kernel it contains information about an external interrupt. Then the related handler is executed.

There are two types of calls. Calls of the first category always return the result directly after they are executed. Examples for these calls are *fork, exec* and *time* (see table 2, entries of this category are marked by *). Calls of the second category may not return directly. They return after the timeout expired, some external interrupt occurred or an other application invoked a related system call. Examples for these calls are *ipcSendto* and *getkeystroke*. In the following we will examine these two examples in more detail.

- **Example SOS IPC**

  SOS provides user applications with an own IPC mechanism (not to confuse with VAMOS IPC). This is because VAMOS IPC is only allowed between the operating system and user applications but not between two user applications. Our architecture is designed in such a way that all communication and services

are exclusively provided by the SOS, this is why SOS has to implement a new communication mechanism.

When an application invokes the system call *ipcSendto*, the SOS first checks whether the sender has permission to send a message to the receiver. If this is the case a new request is enqueued to the request list and the sender will block until either the timeout of this request expires or the receiver invokes the system call *ipcReceive*. Blocking is realized over VAMOS IPC primitive *callService*: the user application is blocking until receiving an answer from the SOS. Else an error is reported in form of a *t_complex* message to the user application (for the exact semantics see [5]).

- **Example getKeyStroke**

  If an user application invokes the system call *getkeystroke* a new request is created and appended to the request list. The application will block until the SOS sends an answer. Meanwhile the SOS goes again into its openwait status, by reinvoking the VAMOS communication primitive *receiveService*. If finally an interrupt occurs indicating that a key was pressed, a message is sent via VAMOS IPC to the SOS, which will then update its data structures and answer the request of the application (for the exact semantics see [5]).

Screen and Keyboard	
lockscreen	requests to lock the screen
unlockscreen*	frees resource screen
setchar	prints a char on the screen
setcursor	moves the cursor screen
lockkeyboard	requests lock of keyboard
unlockkeyboard*	frees the keyboard
getkeystroke	attempts to read a char
Files and more	
createfile	creates a new file
lockfile	locks a file for access
unlockfile*	frees the file
read	attempts to read some bytes of a file
write	attempts to write some bytes of a file
lseek	repositions the offset within a file
Interprocess communication	
ipcSendto	sends a message to an application
ipcRecv	receives a message from any application
ipcSendRecv	sends a message and goes into open receive
And the rest	
fork*	forks a child application
exec*	loads a program from a file
exit*	terminates the application
time*	returns the current SOS time

Table 2: Overview SOS system calls

7.3.2 Handling interrupts

The function $\delta_{interrupt}$ handles local interrupts that occurred in applications. If an external and unpredictable interrupt occurred (indicated in the input variable or in the field *ex*) it reads it and generates a message to the SOS. If no message could be sent because an older interrupt is not yet handled by the SOS the external interrupt is stored in the field *ex*.

$$\delta_{interrupt} : t\_sosstate \longmapsto t\_sosstate$$

The function *isInterrupt* indicates whether a local interrupt occurred (other than trap) or the input signal or the field *ex* is not empty (i.e. external interrupt occurred before ans was not handled yet).

$$isInterrupt : t\_sosstate \times t\_sosinput \longmapsto bool$$

7.3.3 The transition relation

Now we can define the global transition relation. If no interrupt occurs an user application can perform a local step or invokes a system call. The user application is chosen non deterministic. If at least one input variable of the operating system is non empty it chooses the one with the smallest order and applies the semantics of the invoked system call or the semantics of the handler of the reported external (and unpredictable) interrupt. Formally the transition relation is given through:

The expression $[x := c]$ in the left side of the defining equation means, that the result configuration is the given one where the component x is substituted by the value c. The function *getId* returns the ID of an application.

$$\delta_{userapp} : t\_sosstate \times t\_userapp \longmapsto t\_sosstate$$

$$\delta_{userapp}(c, app) = \begin{cases} [c.userapps[getId(app)] := \delta_{local}(app)] & : \text{ if } \overline{isIPC?(app)} \\ \delta_{comm}(c, app) & : \text{ if } isIPC?(app) \end{cases}$$

$$\delta_{sos} : t\_sosstate \times t\_sosinput \longmapsto 2^{t\_sosstate}$$

$$\delta_{sos}(c, i) = \begin{cases} \{\delta_{userapp}(c, c.userapp[k]), \delta_{syscall}(c)\} & : \text{ if } \overline{isInterrupt(c, i)} \wedge \\ & \quad c.userapp[k].alive \\ \{\delta_{interrupt}(c)\} & : \text{ if } isInterrupt(c, i) \end{cases}$$

As mentioned in Section 2 internal transitions $\xrightarrow{\tau}$ can be annotated. We will use the annotations: $syscall^{aid}_{scn}$ if application with ID *aid* invokes the service with number *scn*, $local^{aid}$ if the transition was a local execution of an application with ID *aid* and $usercom^{aid}$ if the transition was the execution of a communication primitive/trap of an application with ID *aid*.

7.4 Correctness

7.4.1 Equivalence

Again we first have to establish a relation between SOS* and the so far built sequence of models. This is done by proving equivalence to the implementation VAMOS*+C0+SOS:

Theorem 9 Equivalence VAMOS*+C0+SOS is equivalent to SOS* under SSIM.

The needed simulation relation SSIM is mainly given through a mapping between the the local state of the SOS in the abstraction and C0 data structures of the implementation. Further the simulation relation maps the states of the abstract devices to the states of the concrete ones. The most challenging part of the proof is the structural abstraction: many interleaved executions become atomic ones. To justify this view, we have to proof that during the execution of the block to be considered atomic, no interleaved execution will interfere. I.e. the execution of the block is *locked* in some way. For that we need important properties of the model. A.o. these are:

- A SOS system call can be considered as one atomic step. In contrast to kernel calls SOS system calls can be interleaved with local computations of applications. These interleaved executions don't interfere with the execution of the system call because of two properties of the SOS. First: the SOS never starts executing a call during the execution of another one. Second: the integrity property of tasks 5.4.2 ensures that the local state of the SOS is only altered through system calls.
- Some system calls involving external devices abstract a complex communication with the hardware and still can be considered as one atomic step. Here we need locks of the hardware given through the communication protocol. For example we suppose that the hard disk will finally (and only then) return a result after the SOS has sent a request.

We further have to prove that the user visible state of a device is consistent with the concrete state of the device. This lemma implicitly includes the correctness of user level device drivers, e.g. the correctness of the file system.

Again it does not suffice only proving equivalence. A number of properties will be necessary for proving correctness of applications.

7.4.2 Availability

Availability says that that all system calls of user applications will finally be received and served by the operating system. That means that the SOS will not lead to starvation of any user processes.

Theorem 10 Availability For all runs (σ, α) of the SOS* model and for all reached states $\sigma[i]$ of the run the following holds:

$$\forall tid : nat.\sigma[i].userapps[tid].comm.out \neq empty \Longrightarrow$$

$$\exists j > i.\sigma[j].userapps[tid].comm.out = empty$$

$$\wedge \sigma[j].userapps[tid].comm.in \neq empty$$

Literally the theorem says, that whenever the output variable of some user application is non empty (i.e. a service is called) after finitely many steps the message is read by the SOS and and answer (either an error message or the result of the call) is sent back. Both reading the message and sending the answer are one step in SOS*.

The proof of this theorem is mainly given through the equivalence theorem. By the definition of SOS* IPC messages with system calls always lead to the execution of the call. The only possible problem here is that the message is not received by the operating system.

7.4.3 Integrity of applications

The integrity theorem for SOS* states that the local state of an application is only altered after local computations or if the application receives some answer to an invoked system call or if another application invokes a *kill* system call and the owner of the application has the privilege to kill.

Theorem 11 Integrity of user applications For all runs (σ, α) of the SOS* model and for all reached states $\sigma[i]$ of the run the following holds:

The constant $const\_SCKILL$ represents the number for the kill system call. $const\_RT\_APP$ stands for the type *application* in resource definitions and $const\_COP\_KILL$ stands for the right to kill a process in a definition of a permission.

$\forall app \in \sigma[i].userapps$ with ID *aid*: $\sigma[i].userapps[aid] \neq \sigma[i+1].userapps[aid] \Longrightarrow$

$$\alpha[i] = local^{aid} \vee$$

$$\alpha[i] = usercom^{aid} \wedge (\sigma[i].userapps[aid].comm.in \neq empty \vee \sigma[i].userapps[aid].localstate = \sigma[i+1].userapps[aid].localstate) \vee$$

$$\alpha[i] = syscall^{cid}_{scn} \wedge (scn = const\_SCKILL \wedge (\exists res1, res2 \in sigma[i].sosds.soslocal.resources : res1.type = const\_RT\_APP \wedge res1.name = cid \wedge res2.type = const\_RT\_APP \wedge res2.name = aid \wedge \exists right \in res2.access : right.type = const\_COP\_KILL \wedge res1.owner \in right.uids)$$

The proof of the integrity theorem for SOS is mainly given through the integrity proof for tasks 5.4.2. One further has to show that the SOS does not manipulate the local states of applications.

7.4.4 Privacy of applications

At this point we would like to shift the privacy theorems (see 5.4.2) from VAMOS* to SOS* and drop the system task condition.[7] Intuitively we will no longer consider *isolated groups of applications* but *isolated groups of users*. For such a group we have to identify all system calls that don't propagate any information to users outside (i.e. applications owned by these users). This is for two reasons a challenging task.

Firstly, most external resources as the keyboard or the monitor are shared by all users. And therefore *locks* of such resources are also visible for *all* users. However, if the applications don't have access to time or SOS cycle information, no arbitrary messages can be encoded through locks. An attacker could only infer if someone in an isolated

[7] Unfortunately we can not use the information flow definitions represented for VAMOS*. That is because the transition relation of SOS* is not deterministic and therefore we can not argue about one run. For non deterministic system information flow could be defined about the concept of non-deductibility.

group is computing or not. Unfortunately system calls in SOS can be specified with timeout values. Through this a complete communication can be established between an application inside the group and one outside.

Secondly, even locks of resources that are non visible outside the group, as files and IPC locks, are maintained for all users in the same SOS data structures. The problem is than given through error messages. E.g. an attacker can try to create a certain file and receive the answer from the SOS that the filename is already used, although *all* information about the file should be invisible for the attacker. Through this simple trick arbitrary messages can be exchanged between applications inside and outside an isolated group. We would need an abstraction of the file system (and the other SOS data structure) establishing some kind of *logical separation*.

As sketched above, some changes have to be made to the SOS to establish a *secure* operating system. However these changes seem not to be fundamentally contrasting the overall concept of the *Simple Operating System*.

7.4.5 Correctness of system calls (IPC)

Similar to VAMOS kernel calls, the correctness of system calls are not only given through equivalence. That is because the specification (or the SOS* model) itself might me *wrong*. Especially for system calls that are not completed directly after invocation, but depend on other events or system calls (as getkeystroke or the SOS IPC mechanism) we have to state correctness properties and verify them in SOS*. We think that for the SOS IPC mechanism the following properties are necessary and needed for higher levels of specifications (e.g. for proving correctness of user applications).

Definition SOS IPC Correctness SOS IPC is correct if the following holds:

- If a rendezvous happens the message that was sent is received without change.
- Only messages that were sent are received.
- Applications waiting for a rendezvous are blocked and never woken up unless the rendezvous can be completed or a specified timeout occurs. In case of a time out all traces of the SOS IPC call are removed.

7.5 Extensions

In this section we outline how new components can be integrated into the framework.

7.5.1 Integration of new operating system functionality

The integration of new system calls into SOS* is straightforward. At first a specification of the system call is formalized in the model. Then the C0 implementation of the call is "plugged into" VAMOS*+C0+SOS. If the new system call does not interact with data structures of the old ones it suffices to prove equivalence between the new semantics and its implementation. Else the entire equivalence theorem has to be reproved.

7.5.2 Integration of new devices

When integrating new devices we will proceed in two steps. At first an abstract *interaction* automaton of the device has to be constructed. Then VAMOS* is instantiated with this new automaton of the device. It can interact via external interrupts and port changes with the kernel.

In the next step we extend the operating system with new system calls providing applications with access functionality to the device. This extension is done at the level of VAMOS*+C0+SOS. The specification of these system calls is given in the model SOS*. Furthermore SOS* introduces a new, more abstract model of the device (only the user visible state of the machine).

Now we prove the correctness of the specification against the C0 implementation of the SOS system calls. At this point also the correctness of our new abstraction of the device is verified. In particular a correspondence between the user visible state and the state of the abstract interaction automaton of the device is constructed. The specification of the system call can hide away a complex communication protocol with the device automaton.

7.5.3 Integration of servers

A server is a concrete application that provides some kind of services to other applications. It is integrated into the operating system environment in two steps. First we *instantiate* SOS* with the server (given as C0 machine) we want to build in to the model SOS*+server. Then we *abstract* SOS*+server to a model SOS+server* in which only a specification of the interaction between server, operating system and the other applications is visible. This specification is verified against the C0-implementation.

8 Summing Up

8.1 Following the road

In this thesis we first tried to figure out, how different models are related to each other and what we understand under correctness of models. Two main techniques for building new models out of old ones were identified: *instantiating* and *abstraction*. Although these terms and notions are often used in a self-evident way, defining them formally turned out to be a challenging task.

After gaining insight into the relationship between models of different layers, we were able to form and ascend the *verification stairs*. The scheme of instantiating parameters of underlying models, and then simplifying these new models with the help of different types of abstractions, shaped up as being a very easy and effective way for putting together all so far implemented and modeled layers in the academic system. Meanwhile the *verification stairs* have succeeded in the *Verisoft* project, as it has been also used for describing the modeling process of OsekTime/Flexray.

Indeed only two forms of abstraction were used. Firstly, so called primitives served as code abstraction. Primitives were used in all models: CVM* primitives abstracting Assembler instructions, kernel calls, abstracting code consisting of C0 Statements and CVM* primitives, communication primitives, abstracting the trap instruction and finally system calls, abstracting code consisting of C0 Statements and kernel calls. Secondly, structural abstraction was used to simplify communication processes, e.g. representing IPC as shared variables.

Step-by-step implementations of all layers have been integrated in the framework. A main aspect of this process was introducing C0 semantics for user applications in VAMOS*. By that a real concurrent model was established into which the *Simple Operating System* could be integrated. Furthermore modeling and abstracting IPC mechanisms in a consistent way, introducing external devices and a uniform handling of the interrupt mechanism were crucial while joining different models.

The main correctness criteria we identified in each layer were equivalence to the underlying model, functional correctness of calls (kernel calls and system calls), integrity and privacy of processes, tasks and applications. Especially privacy of tasks in the μ-kernel VAMOS turned out to be a strong property.

8.2 Further work

We only gave arguments why the stated theorems for each model *probably* hold. Detailed proofs are still missing. We are also aware that the set of theorems is far from being complete. Some needed properties (especially properties of the different calls, i.e. functional correctness) may only become visible in higher layers, e.g. when applications in SOS* are verified.

Some parts of the framework have been implemented as part of the Verisoft project in the higher order proof assistant system *Isabelle*. These models are CVM* [1, 15] and partially VAMOS* (nothing published yet). We still need *Isabelle* implementations of VAMOS*+C0 and SOS*. Furthermore all stated theorems have to be formulated and proven in *Isabelle*.

As described in the last section we could not lift the privacy theorem to SOS*. For this some changes in the design have to be done. In addition the SOS should be instantiated with some basic servers. Furthermore description of most basic abstract devices (in form of finite state automata) is still missing. In [14] a.o. the modeling and integration of the hard disk is described.

8.3 Conclusion

We sketched a *road map* for modeling and verifying a pervasive system of communicating applications in operating system environment. A complete specification of the framework beginning with an abstract processor model, reaching the kernel layer and finally the operating system layer, could be described formally in less than 50 pages. Now assertions over the models can be stated unambiguously and proven, and on top of SOS* new applications can be implemented and verified.

References

[1] Eyad Alkassar. Communicating Virtual Machines. Formale Spezifikation von Konfiguration und Funktionalität. Bachelor thesis, 2005.

[2] Christoph Berg, Michael Klein, Dirk Leinenbach, and Wolfgang Paul. Formal Operational Semantics of C0. ITB#8, not yet published, 2004.

[3] William R. Bevier, Warren A. Hunt Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.

[4] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together - formal verification of the VAMP. *In STTT Journal, Special Issue on Recent Advances in Hardware Verification, Springer*, 15(11), (to appear) 2005.

[5] Sebastian Bogan. A formal specification of the simple operating system. ITB#50, not yet published, 2005.

[6] Sebastian Bogan. Implementation and model of sos system calls. ITB#29, not yet published, 2005.

[7] The Verisoft Consortium. The Verisoft Project. http://www.verisoft.de/, 2003.

[8] Mauro Gargano, Dirk Leinenbach, Mark Hillebrand, and Wolfgang Paul. Simulating CVM machines by physical DLX machines. ITB#30, not yet published, 2004.

[9] Mauro Gargano, Dirk Leinenbach, and Wolfgang Paul. CVM: Communicating Virtual Machines. ITB#10, not yet published, 2005.

[10] Mauro Gargano and Wolfgang Paul. A CVM specialization: The VAMOS operating system kernel. ITB#38, not yet published, 2005.

[11] J.A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. Security and Privacy*, pages 11–20, 1982.

[12] Matthew Hennessy. *Algebraic theory of processes*. MIT Press, Cambridge, MA, USA, 1988.

[13] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctnesss*. PhD thesis, Saarland U, Saarbrücken, Germany, 2005.

[14] Mark A. Hillebrand, Thomas In der Rieden, and Wolfgang J. Paul. Dealing with I/O devices in the context of pervasive system verification. not yet published, 2005.

[15] Steffen Knapp. Communicating Virtual Machines. Dynamischer Teil. Bachelor thesis, 2005.

[16] Dirk Leinenbach. Die Sprache C0. ITB#2, not yet published, 2005.

[17] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Compiler verification in the context of pervasive system verification. not yet published, 2005.

[18] J. Liedtke. *μ-Kernels Must And Can Be Small*. 5th IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS), Seattle, WA, 1996.

[19] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[20] J. Strother Moore. A grand challenge proposal for formal methods: A verified stack. In *10th Anniversary Colloquium of UNU/IIST*, pages 161–172, 2002.

[21] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.

[22] Tutorial and introduction to Isabelle. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[23] Open systems and the corresponding interfaces for automotive electronics. http://www.osek-vdx.org/, 2005.

[24] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[25] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. In *Proceedings of the International Workshop on Test and Analysis of Component-based Systems (TACoS'04)*, 2004.

[26] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

[27] Werner Stephan. Concurrency - The Semantic Model. ITB#24, not yet published, 2004.

List of Figures

List of Tables