Assumption-Commitment Types for Resource Management in Virtually Timed Ambients

Einar Broch Johnsen, Martin Steffen, and Johanna Beate Stumpf

University of Oslo, Oslo, Norway
{einarj,msteffen,johanbst}@ifi.uio.no

Abstract. This paper introduces a type system for resource management in the context of nested virtualization. With nested virtualization, virtual machines compete with other processes for the resources of their host environment in order to provision their own processes, which could again be virtual machines. The calculus of virtually timed ambients formalizes such resource provisioning, extending the capabilities of mobile ambients to model the dynamic creation, migration, and destruction of virtual machines. The proposed type system is compositional as it uses assumptions about the outside of a virtually timed ambient to guarantee resource provisioning on the inside. We prove subject reduction and progress for well-typed virtually timed ambients, expressing that upper bounds on resource needs are preserved by reduction and that processes do not run out of resources.

1 Introduction

Virtualization enables the resources of an execution environment to be represented as a software layer, a so-called *virtual machine.* Software processes are agnostic to whether they run on a virtual machine or directly on physical hardware. A virtual machine is itself such a process, which can be executed on another virtual machine. Technologies such as VirtualBox, VMWare ESXi, Ravello HVX, Microsoft Hyper-V, and the open-source Xen hypervisor increasingly support running virtual machines inside each other in this way. This *nested virtualization*, originally introduced by Goldberg [1], is necessary to host virtual machines with operating systems which themselves support virtualization [2], such as Microsoft Windows 7 and Linux KVM. Use cases for nested virtualization include end-user virtualization for guests, software development, and deployment testing. Nested virtualization is also a crucial technology to support the hybrid cloud, as it enables virtual machines to migrate between different cloud providers [3].

To study the logical behavior of virtual machines in the context of nested virtualization, this paper introduces a type-based analysis for a calculus of virtual machines. An essential feature of virtual machines, captured by this calculus, is that a virtual machine competes with other processes for the resources available in their execution environment, in order to provision resources to the processes *inside* the virtual machine. Another essential feature of virtual machines is *migration.* From an abstract perspective, virtual machines can be seen as mobile processes which can move between positions in a hierarchy of nested locations.

We develop the type system for *virtually timed ambients* [4], a calculus of mobile virtual locations with explicit resource provisioning, based on mobile ambients [5]. The goal is to statically approximate an upper bound on resource consumption for systems of virtual machines expressed in this calculus. The calculus features a resource called *virtual time*, reflecting local execution capacity, which is provisioned to an ambient by its parent ambient, similar to time slices that an operating system provisions to its processes. With several levels of nested virtualization, virtual time becomes a *local* notion which depends on an ambient's position in the location hierarchy. Virtually timed ambients are mobile, reflecting that virtual machines may migrate between host virtual machines. Migration affects the execution speed of processes inside the virtually timed ambient which is moving as well as in its host before and after the move. Consequently, the resources required by a process change dynamically when the topology changes.

The distinction between the inside and outside of a virtually timed ambient (or a virtual machine) is a challenge for compositional analysis; we have knowledge of the current contents of the virtual machine, but not of what can happen outside its borders. This challenge is addressed in our type system by distinguishing *assumptions* about ambients on the outside of the virtually timed ambient from *commitments* to ambients on the inside. To statically approximate the effects of migration, an ambient's type imposes a bound on the ambients it can host. If type checking fails, the ability to provision resources for an incoming ambient in a timely way cannot be statically guaranteed in our type system.

The ambient calculus has previously been enriched with types (e.g., [6]). Exploiting the explicit notion of resource provisioning in virtually timed ambients (including a fair resource distribution and competition for resources between processes), our type system captures the resource capacity of a virtually timed ambient and an upper bound on the number of its subambients. The type system thereby provides concrete results on resource consumption in an operational framework. Resource dependency in the type system is expressed using *coeffects.* The term coeffect was coined by Petricek, Orchard, and Mycroft [7, 8] to capture how a computation depends on an environment rather than how it affects the environment. In our setting, coeffects capture how a process depends on its environment by an upper bound on the resources needed by the process.

Contributions. The main technical contributions of this paper are

– an *assumption commitment type system* with *effects* and *coeffects*, which provides a static approximation of constraints regarding the capacity of virtually timed ambients and an upper bound on their resource usage; and
– a proof of the *soundness of resource management* for well-typed virtually timed ambients in terms of a *subject reduction* theorem which expresses that the upper bounds on resources and on the number of subambients are preserved under reduction, and a *progress* theorem which expresses that well-typed virtually timed ambients will not run out of resources.

To the best of our knowledge, this is the first assumption commitment style type system for resource types and nested locations.

Paper overview. Section 2 introduces virtually timed ambients. Section 3 presents the type system for resource management. In Section 4, we prove the soundness of the type system in terms of subject reduction and progress. We discuss related work and conclude in Sections 5 and 6.

2 Virtually Timed Ambients

Mobile ambients [5] are processes with a concept of location, arranged in a hierarchy which may change dynamically. Interpreting these locations as places of deployment, *virtually timed ambients* [4,9] extend mobile ambients with notions of virtual time and resource consumption. The timed behavior of a process depends on the one hand on the *local* timed behavior, and on the other hand on the placement or deployment of the process in the hierarchical ambient structure. Virtually timed ambients combine timed processes and timed capabilities with the mobility and location properties of the mobile ambient calculus.

Compared to Johnsen *et al.* [4,9], we here present a slightly simplified version of virtually timed ambients which assumes a uniform speed for all ambients in the hierarchy. This simplification does not mean the ambients proceed uniformly with respect to time: the progress of an ambient still depends on its position in the hierarchy and the number of sibling ambients that compete for time slices at the given level. When discussing the reduction rules from Table 1 later, we provide further details on how the more general non-uniform setting relates to the presentation here. Since an ambient system can change its structure, i.e., its hierarchy, an ambient's local access to time slices may also dynamically change. Thus, the simplification by uniform speed is not conceptual, but it allows a simpler formulation of the type system by removing fractional representations of speed in scheduling and the resulting (easy but cumbersome) calculations.

Definition 1 (Virtually timed ambients). *The syntax of virtually timed ambients is as follows:*

$$\begin{aligned} P &::= \mathbf{0} \mid (\nu n)\, P \mid P|P \;\mid\, !C.P \;\mid C.P \mid n[P] \\ C &::= \mathbf{in}\, n \mid \mathbf{out}\, n \mid \mathbf{open}\, n \mid \mathbf{c} \end{aligned}$$

The syntax is almost unchanged from that of standard mobile ambients (e.g., [5]); the only syntactic addition is an additional capability **c** explained below. In the sequel, we mostly omit the qualification "timed" or "virtually timed" when speaking about processes, capabilities, etc. Processes include the inactive process **0**, parallel composition $P \mid P$ and replication $!C.P$, the latter conceptually represents an unbounded parallel composition of a process, with capability C as "guard". The ν-binder or restriction operator, makes the name n local, as in the π-calculus, ambient calculus, and related formalisms. *Ambients* $n[P]$ are named processes. The standard mobile ambient *capabilities* **in**, **out**, and **open** allow a process to change the nested ambient structure by moving an ambient into or out of another ambient, or by dissolving an ambient altogether.

The additional capability $\mathtt{c}$ is specific for the virtually timed extension and abstractly represents the need of the process for a *resource* in order to continue its execution (i.e., $\mathtt{c}$ can be read as "consume"). Thus, the consume capability relates to resource cost in frameworks for cost analysis (e.g., [10, 11]). In our setting, the $\mathtt{c}$-capabilities consume resources which can be thought of as time slices and which are governed by a scheduler. A scheduler is *local* to an ambient and its responsibility is to schedule resources to the processes directly contained in this ambient. Since ambients are nested, the scheduler also has to allocate time slices or resources to subambients, thereby delegating the allocation of time slices at the level of the subambients to their respective schedulers. To achieve a fair distribution of resources, the semantics adopts a simple round-based scheduling strategy. Intuitively, no process is served twice, unless all other processes that need resources at that level have been served at least once. This *round-based scheme* is slightly more refined in that the number of processes per ambient is not fixed as ambients may move inside the hierarchy and even dissolve.

To capture the outlined scheduling strategy in operational rules working on the syntax of ambients, we *augment* the grammar of Def. 1 with additional *run-time* syntax (highlighted below). When needed, we refer to the original syntax from Def. 1 as *static* syntax. The run-time syntax uses the notation $\check{\ }$ to indicate that processes, including ambients, are *frozen* and $\overline{n}$ to denote either n or $\check{n}$.

$$\begin{aligned}
P &::= \mathbf{0} \mid (\nu n)\, P \mid \ P \mid P \ \mid !C.P \mid \ \boxed{\mathtt{tick}?} \mid \boxed{\mathtt{tick}!} \mid \overline{n}[P] \mid C.P \\
\overline{n} &::= n \mid \boxed{\check{n}} \\
\gamma &::= \mathtt{c} \mid \boxed{\check{\mathtt{c}}} \\
C &::= \mathtt{in}\, n \mid \mathtt{out}\, n \mid \mathtt{open}\, n \mid \boxed{\mathtt{tick}?} \mid \gamma
\end{aligned}$$

Frozen processes are not eligible for scheduling. For regular (non-ambient) processes, only processes prefixed by the consume capability $\mathtt{c}$ will be controlled in this way; other processes are unconditionally enabled. Consequently, we only need as additional run-time syntax $\check{\mathtt{c}}$, capturing a deactivated resource capability. Similarly $\check{n}[P]$ denotes a timed ambient which is not eligible for scheduling. Apart from scheduling, a frozen ambient $\check{n}[P]$ is treated as any other ambient $n[P]$: the ordinary, untimed capabilities address ambients by their name without the additional scheduling annotation. Likewise, ν-binders and corresponding renaming and algebraic equivalences treat names $\check{n}$ as identical to n. Unless explicitly mentioned, we assume in the following run-time syntax, i.e., P may contain occurrences of $\check{n}$ and $\check{\mathtt{c}}$. Time slices are denoted by ticks, and come in two forms $\mathtt{tick}?$ and $\mathtt{tick}!$. We may think of the first form $\mathtt{tick}?$ as representing incoming ticks into an ambient, typically from the parent ambient, the second form $\mathtt{tick}!$ represents time slices handed out to the local processes by the local scheduler. The $\mathtt{tick}?$-capability similarly accepts an incoming tick. Let $names(P)$ denote the set of names for ambients contained in P.

Semantics. The semantics of virtually timed ambients is given as a *reduction* relation $P \rightarrowtriangle Q$ (see Tables 1 and 2). Processes are considered up-to structural congruence $P \equiv Q$ and reduction is defined modulo $\equiv$. The corresponding rule

(R-In)	$\overline{n}[\mathbf{in}\, m.P_1 \mid P_2] \mid \overline{m}[Q] \rightharpoonup \overline{m}[Q \mid \check{n}[P_1 \mid P_2]]$
(R-Out)	$\overline{m}[\overline{n}[\mathbf{out}\, m.P_1 \mid P_2] \mid Q] \rightharpoonup \check{n}[P_1 \mid P_2] \mid \overline{m}[Q]$
(R-Open)	$\mathbf{open}\, n.P_1 \mid \overline{n}[P_2] \rightharpoonup P_1 \mid \check{P}_2$

Table 1. Reduction rules (1). The symbol $\bar{m}$ occuring both on the left and the right side of a reduction rule represents either $\check{m}$ on both sides, or else m on both sides.

is omitted here, as is the standard axiomatization of $P \equiv Q$. We further omit the standard congruence rules (e.g., $P \rightharpoonup P' \Rightarrow \overline{n}[P] \rightharpoonup \overline{n}[P']$), which also correspond to those for mobile ambients. The rules in Table 1 (with rule names to the left) cover ambient reconfiguration. Apart from the annotations used for scheduling, the rules are exactly the ones for the (untimed) mobile ambients [5].

Ambients can undergo restructuring in three different ways. First, an ambient can move horizontally or laterally by entering a sibling ambient (rule R-In). Second, it can move vertically up the tree, leaving its parent ambient (rule R-Out). Finally, a process can cause the dissolution of its surrounding ambient (rule R-Open). These forms of restructuring are *untimed* in that they incur no resource cost. If an ambient changes its place, the scheduler of the target ambient will, from that point on, become responsible for the new ambient, and the treatment is simple: Being frozen, the newcomer will not be served in the current round of the scheduler, but waits for the next round. Considering the source ambient (i.e., the ambient which contained the process executing the **out** or **in** capability), no process inside the source ambient looses or changes its status. A similar discipline is followed (for P_1) when opening an ambient in rule R-Open (the interpretation of $\widehat{P}_2$ as opposed to P_2 follows shortly). Note that a process in an ambient can execute a capability **in**, **out**, or **open** independent of the status of the affected ambient, which is indicated in the rules by $\overline{n}$ and $\overline{m}$.

Scheduling, in particular the handling of ticks and the resource capabilities, is covered by the reduction rules in Table 2. Scheduling ultimately means to discriminate and select between processes which are allowed to proceed at a given point, and those which are not. To capture that distinction in the rules, $\widehat{P}$ represents the former, i.e. P is eligible for a new time-slice ("unfrozen"). Dually, $\check{P}$ represents P after having been served ("frozen"). The exact formulation of the freeze and unfreeze operation will be given in Def. 3 below, after discussing the rules themselves. The first rule translates "incoming" ticks to ticks available for local processes. The translation ratio is uniform; i.e., one incoming tick produces one outgoing tick (this is the simplification compared to previous work mentioned earlier, where the ratio between incoming and local ticks could more generally be a rational number). A **tick**! process can be consumed in two ways. First by scheduling a **c**-prefixed process which undergoes the steps $\mathbf{c}.P \rightharpoonup \mathbf{tick}?.P \rightharpoonup \check{P}$ (consuming **tick**! in the second step).[1] Second, by scheduling a subambient,

[1] The rules and the calculus may be simplified, e.g., by avoiding the two-step behavior just described. The formulation here was chosen as it more aligned with versions of

$$
\begin{array}{lll}
\texttt{tick}? \rightarrowtriangle \texttt{tick}! & \texttt{tick}! \mid \texttt{tick}?.P \rightarrowtriangle \check{P} & \dfrac{\text{not}(P{\downarrow_{\texttt{tick}?}})}{\overline{n}[P] \rightarrowtriangle \overline{n}[\widehat{P}]} \\
\texttt{c}.P \rightarrowtriangle \texttt{tick}?.P & \texttt{tick}! \mid n[P] \rightarrowtriangle \check{n}[\texttt{tick}? \mid P] &
\end{array}
$$

Table 2. Reduction rules (2)

such that an incoming tick $\texttt{tick}?$ occurs one level down in the hierarchy. To ensure the round-based scheduling, the scheduled entity must not have been served yet in the current round. For this purpose, the process before the transition must be of the form $\texttt{tick}?.P$ or $n[P]$, and after the transition the continuation of the process is frozen, using Def. 3. The last rule completes one scheduling round and initiates the next round by changing the ambient's processes P to $\widehat{P}$. This unfreezing step can be done only if all the ambient's processes have been served, which is captured be the rule's negative premise, stipulating that no process at the level of n can proceed.

This inability to proceed by a tick-step at the end of a round is formulated using the notion of *observables,* also known as *barbs.* Barbs, originally introduced for the π-calculus [12], capture a notion of immediate observability. In the ambient calculus, these observations concern the presence of a top-level ambient whose name is not restricted [13]. In our context, the barbs are adapted to express *top-level* schedulability, i.e., an ambient's ability to receive a tick. Later, to formulate progress properties, we will additionally need to capture the same condition for t a sub-process deeper inside the system and not necessarily at top-level. For that, we denote by $\mathcal{C}[\cdot]$ (or simply by $\mathcal{C}$) a *context*, i.e., a process with a (unique) hole $[\cdot]$ in place of a process, and write $\mathcal{C}[P]$ for the context with its hole replaced by P (for the formal definition, see [13]). The observability predicates (or "tick-barbs") $\downarrow_{\texttt{tick}?}$ resp. $\downarrow^{\mathcal{C}}_{\texttt{tick}?}$ are then defined as follows, where $\widetilde{m}$ is a tuple of names:

Definition 2 (Barbs). *A process P has a* strong barb on $\texttt{tick}?$*, written $P{\downarrow_{\texttt{tick}?}}$, if $P \equiv (\nu\widetilde{m})(n[P_1] \mid P_2)$ or $P \equiv (\nu\widetilde{m})(\texttt{tick}?.P_1 \mid P_2)$. A process P* has a strong barb on $\texttt{tick}?$ in context $\mathcal{C}$*, written $P{\downarrow^{\mathcal{C}}_{\texttt{tick}?}}$, if $P = \mathcal{C}[P']$ for some process P' with $P'{\downarrow_{\texttt{tick}?}}$.*

Note that the ambient name n may well be hidden, i.e., mentioned in $\widetilde{m}$. Barbing on the ambient name n, written $P{\downarrow_n}$, would require that $P \equiv (\nu\widetilde{m})(n[P_1] \mid P_2)$ where $n \notin \widetilde{m}$, in contrast to the definition of $P{\downarrow_{\texttt{tick}?}}$. This more conventional notion of strong barbs [13] expresses that an ambient is available for interaction with the standard ambient capabilities; ambients whose name is unknown are not available to be contacted by other ambients and therefore, their name is excluded in the observability predicate $\downarrow_n$. In contrast, strong barbs as

virtually timed ambients allowing non-uniform speeds across ambients, mentioned earlier in this section. Thus, the type system here would allow a more straightforward generalization to ambients with non-uniform speed.

defined in Def. 2 capture an ambient's ability to receive ticks and thus, the definition will allow hidden ambients to be served by the local scheduler. However, the name of the ambient must *not* be frozen $\check{n}$: ambients that have been served a tick in the current round are not eligible for another allocation before a new round has started, in which case the ambient's name has "changed" to n.

To complete the presentation of the semantics, we provide the operations used in the rules that allow processes to conceptually switch back and forth between waiting to be served in the current round, and having been served and thus waiting for the next round to begin.

Definition 3 (Freezing and unfreezing). *Let $\widecheck{P}$ denote the process where all top-level occurrences of $n[Q]$ are replaced by $\check{n}[Q]$ and all top-level occurrences of $\mathbf{c}$ replaced by $\check{\mathbf{c}}$. Conversely, let $\widehat{P}$ denote the process where all top-level occurrences of $\check{\mathbf{c}}$ are replaced by $\mathbf{c}$ and all top-level occurrences of $\check{n}[Q]$ replaced by $n[Q]$. Define $\widecheck{P}$ by induction on the syntactic structure as follows:*

$$
\begin{array}{rcll}
\widecheck{(\nu n)\, P} &=& (\nu n)\, \widecheck{P} & \qquad\qquad \check{\check{\mathbf{c}}} = \check{\mathbf{c}} \\
\widecheck{P_1 \mid P_2} &=& \widecheck{P_1} \mid \widecheck{P_2} & \qquad\qquad \check{\mathbf{c}} = \check{\mathbf{c}} \\
\widecheck{n[P]} &=& \check{n}[P] & \qquad\qquad \check{n} = \check{n} \\
\widecheck{\gamma.P} &=& \check{\gamma}.P & \qquad\qquad \check{\check{n}} = \check{n} \\
\widecheck{C.P} &=& C.\widecheck{P} & C \nparallel \gamma \\
\widecheck{P} &=& P & \textit{otherwise}
\end{array}
$$

The definition of $\widehat{P}$ is analogous (e.g., $\hat{\check{\mathbf{c}}} = \mathbf{c}$) and omitted here.

Remark that the congruence relation, which is part of the reduction semantics, works with scheduling in the sense that both operations defined in Def. 3 are preserved under congruence: $P_1 \equiv P_2$ implies $\widecheck{P_1} \equiv \widecheck{P_2}$ and $\widehat{P_1} \equiv \widehat{P_2}$.

Example 1. Consider the process $\mathit{cloud}\,[\mathbf{0}] \mid \mathit{vm}[\mathbf{in}\ \mathit{cloud}.\mathbf{c}.\mathbf{0}]$. If we place this process in a context *root* with one **tick**! process, three reduction steps become possible, as **tick**! can propagate to either ambients and ambient *vm* can move into *cloud*. One way this process can reduce, is as follows:

$$
\begin{array}{l}
\mathit{root}[\mathbf{tick}! \mid \mathit{cloud}\,[\mathbf{0}] \mid \mathit{vm}[\mathbf{in}\ \mathit{cloud}.\mathbf{c}.\mathbf{0}]] \rightarrowtriangle \mathit{root}[\mathbf{tick}! \mid \mathit{cloud}\,[\mathbf{0} \mid \check{\mathit{vm}}[\mathbf{c}.\mathbf{0}]]] \\
\quad \rightarrowtriangle \mathit{root}[\check{\mathit{cloud}}\,[\mathbf{tick}? \mid \mathbf{0} \mid \check{\mathit{vm}}[\mathbf{c}.\mathbf{0}]]] \rightarrowtriangle \mathit{root}[\check{\mathit{cloud}}\,[\mathbf{tick}! \mid \mathbf{0} \mid \mathit{vm}[\mathbf{c}.\mathbf{0}]]] \\
\quad \rightarrowtriangle \mathit{root}[\check{\mathit{cloud}}\,[\mathbf{0} \mid \check{\mathit{vm}}[\mathbf{tick}? \mid \mathbf{c}.\mathbf{0}]]] \rightarrowtriangle \mathit{root}[\check{\mathit{cloud}}\,[\mathbf{0} \mid \check{\mathit{vm}}[\mathbf{tick}! \mid \mathbf{c}.\mathbf{0}]]] \\
\quad \rightarrowtriangle \mathit{root}[\check{\mathit{cloud}}\,[\mathbf{0} \mid \check{\mathit{vm}}[\mathbf{tick}! \mid \mathbf{tick}?.\mathbf{0}]]] \rightarrowtriangle \mathit{root}[\check{\mathit{cloud}}\,[\mathbf{0} \mid \check{\mathit{vm}}[\mathbf{0}]]]\ .
\end{array}
$$

However, the time slice could also enter the ambient *vm*, and move with this ambient, resulting in a reduction sequence starting as follows:

$$
\begin{array}{l}
\mathit{root}[\mathbf{tick}! \mid \mathit{cloud}\,[\mathbf{0}] \mid \mathit{vm}[\mathbf{in}\ \mathit{cloud}.\mathbf{c}.\mathbf{0}]] \\
\quad \rightarrowtriangle \mathit{root}[\mathit{cloud}\,[\mathbf{0}] \mid \check{\mathit{vm}}[\mathbf{tick}? \mid \mathbf{in}\ \mathit{cloud}.\mathbf{c}.\mathbf{0}]] \\
\quad \rightarrowtriangle \mathit{root}[\mathit{cloud}\,[\mathbf{0} \mid \check{\mathit{vm}}[\mathbf{tick}? \mid \mathbf{c}.\mathbf{0}]]] \rightarrowtriangle \ldots
\end{array}
$$

Generally, a process P will be placed in a runtime environment which provisions it with a given amount of resources (e.g., *root* in Ex. 1 with one **tick**!).

When executed in a surrounding ambient without enough resources, some sub-process of P may not receive a sufficient number of resources and may get "stuck". This inability to progress for lack of resources can be captured by (contextually) having a barb on an irreducible process: $P \not\rightarrowtriangle$ and $P{\downarrow}^{\mathcal{C}}_{\texttt{tick?}}$ for some $\mathcal{C}$, i.e., P cannot proceed despite the fact that there is a sub-process that could proceed by consuming a resource, if one were still available. This intuition is used to formulate progress (Theorem 2), stipulating that well-typed processes will not get stuck.

3 An Assumption-Commitment Type System

We consider a type system which analyzes the timed behavior of virtually timed ambients in terms of the movement and resource consumption of a given process. Statically estimating the timed behavior is complicated because the placement of an ambient in the process hierarchy influences its resource consumption, and movements inside the hierarchy change the relative speed of the ambients. The proposed type system is loosely based on Cardelli, Ghelli, and Gordon's movement control types for mobile ambients [14]; however, its purpose is quite different, and therefore the technical formulation will be rather different as well.

Types and typing contexts. Processes will be typed with respect to nominal resource contracts for virtually timed ambients, which are tuples of the form

$$T = \langle \mathit{cap}, \mathit{bnd}, \mathit{tkn} \rangle.$$

Here, $\mathit{cap} \in \mathbb{N}$ specifies the ambient's *resource capacity*, i.e., the upper bound on the number of resources that the subprocesses of the ambient are allowed to require; $\mathit{bnd} \in \mathbb{N}$ specifies the ambient's *hosting capacity*, i.e., the upper bound on the number of timed subambients and timed processes allowed inside this ambient; and $\mathit{tkn} \in \mathbb{N}$ specifies the ambient's *currently hosted processes*, i.e., the number of taken slots within the ambient's hosting capacity. The number of currently hosted processes inside an ambient can change dynamically, due to the movements of ambients. These changes must be captured in the type system. In this sense, a type for ambient names T contains an accumulated effect mapping.

Typing *environments* or contexts associate ambient names with resource contracts. They are finite lists of associations of the form $n : T$. In the type system, when analyzing an ambient or process, a typing environment will play a role as an *assumption*, expressing requirements about the ambients *outside* the current process. Dually, facts about ambients which are part of the current process are captured in another typing environment which plays the role of a *commitment.* Notationally, we use Γ for assumption and Δ for commitment environments. We write $\varnothing$ for the empty environment, and $\Gamma, n : T$ for the extension of Γ by a new binding $n : T$. We assume that ambient names n are unique in environments, so n is not already bound in Γ. Conversely, $\Gamma \backslash n : T$ represents an environment coinciding with Γ except that the binding for n is removed. If n is not declared in Γ, the removal has no effect. The typing judgment for names is given as

$\Gamma \vdash n : T$. Since each name occurs at most once, an environment Γ can be seen as a finite mapping; we use $\Gamma(n)$ to denote the ambient type associated with n in Γ and write $dom(\Gamma)$ for all names bound in Γ. In the typing rules, the typing environment Γ may need to capture the ambient in which the current process resides; this ambient will conventionally be denoted by the *reserved name* **`this`**.

We now define domain equivalence, context addition, error-free environments, and an ordering relation on types and environments to capture subtyping.

Definition 4 (Domain equivalence). *Two contexts Γ_1 and Γ_2 are* domain equivalent*, denoted $\Gamma_1 \sim \Gamma_2$, iff $dom(\Gamma_1) = dom(\Gamma_2)$.*

Definition 5 (Additivity of contexts). *Let Γ_1 and Γ_2 be contexts such that $\Gamma_1 \sim \Gamma_2$, and $\Gamma_i(n) = \langle cap, bnd, tkn_i \rangle$ for $n \in dom(\Gamma_1)$ and $i = 1, 2$. The context $\Gamma_1 \oplus \Gamma_2$ with domain $dom(\Gamma_1)$ is defined as follows: for $n \in dom(\Gamma_1)$*

$$(\Gamma_1 \oplus \Gamma_2)(n) = \langle cap, bnd, tkn_1 + tkn_2 \rangle.$$

If the number of currently hosted ambients is smaller than the hosting capacity of all ambients in an environment, we say that the environment is error-free:

Definition 6 (Error-free environments). *An environment Γ is* error-free*, denoted $\vdash \Gamma : \mathbf{ok}$ if $tkn \leqslant bnd$ for all $n \in dom(\Gamma)$ and $\Gamma(n) = \langle cap, bnd, tkn \rangle$.*

Resource contracts can be ordered by their contents and environments by their resource contracts. The bottom type $\bot$ is a subtype of all resource contracts.

Definition 7 (Ordering of resource contracts and environments). *Let $T_1 = \langle cap_1, bnd_1, tkn_1 \rangle$ and $T_2 = \langle cap_2, bnd_2, tkn_2 \rangle$ be resource contracts. Then T_1 is a* subtype *of T_2, written $T_1 \leqslant T_2$, if and only if $cap_1 \leqslant cap_2$, $bnd_1 \leqslant bnd_2$ and $tkn_1 \geqslant tkn_2$. Typing environments Γ_1 and Γ_2 are* ordered *by the subtype relation as follows: $\Gamma_1 \sqsubseteq \Gamma_2$ if and only if $dom(\Gamma_1) \subseteq dom(\Gamma_2)$ and $\Gamma_1(n) \leqslant \Gamma_2(n)$, for all $n \in dom(\Gamma_1)$.*

Typing judgments. A typing judgment for a process P has the form

$$\Gamma;\ req \vdash P : \mathbf{ok}\langle prov, subs \rangle;\ \Delta$$

where *req* and *prov* are the required and provided resources for P, *subs* is the number of subambients of P, and Γ and Δ are the assumptions and commitments of P, respectively. Scheduling is reflected in the type rules by the calculation of the required resources *req*, which capture the number of resources a process will need to make progress. We call *req* the *coeffect* of the process. Coeffects [7, 8] capture how a computation depends on an environment rather than how it affects this environment. We use the perspective of coeffects since a computation may require resources from its environment to terminate. Similarly, *prov* is the number of provided resources in P; these resources are available in P independent of its environment, and *subs* approximates the number of subambients in P. We may think of $\langle prov, subs \rangle$ as the *effect* of the typing judgment, where effects express what the process P potentially provides to its environment.

For each process, the domain of the assumptions is assumed to contain all names which are not in the domain of the commitments; i.e., for two parallel processes P_1 and P_2 such that $\mathit{\Gamma}_1; \mathit{req}_1 \vdash P_1 : \mathbf{ok}\langle \mathit{prov}_1, \mathit{subs}_1 \rangle; \mathit{\Delta}_1$ and $\mathit{\Gamma}_2; \mathit{req}_2 \vdash P_2 : \mathbf{ok}\langle \mathit{prov}_2, \mathit{subs}_2 \rangle; \mathit{\Delta}_2$, we will have that $\mathit{\Delta}_2 \subseteq \mathit{\Gamma}_1$, $\mathit{\Delta}_1 \subseteq \mathit{\Gamma}_2$ and $\mathit{dom}(\mathit{\Delta}_1) \cap \mathit{dom}(\mathit{\Delta}_2) = \varnothing$. Since ambient names are assumed to be unique, it follows for type judgments that $\mathit{dom}(\mathit{\Delta}) \cap \mathit{dom}(\mathit{\Gamma}) = \varnothing$, as an ambient is either inside the process and has its contract in the commitments, or outside and has its contract in the assumptions. Further, $\mathit{dom}(\mathit{\Delta}) \subseteq \mathit{names}(P)$.

In Table 3, Rule T-Zero types the inactive process, which does not require nor provide any time slices. Rule T-Tick1 expresses the availability of **tick**! and Rule T-Tick2 that a time slice **tick**? is ready to be consumed. Both judgments express that a time slice is provided without requiring any time slice. The assumption rule T-Ass types an ambient with the resource contract it has in the environment. The restriction rule T-Res removes the resource contract assumption in the environment for the restricted name. Subsumption relates different resource contracts; e.g., in subtypes (T-Tsub), the subsumption rule T-Sub allows a higher number of required resources, a lower number of provided resources and a higher number of subambients to be assumed in a process.

For the typing of ambients in Rule T-Amb, the reserved name **this** is used to denote the *current environment* of P in the premise of the rule; the assumed typing of **this** becomes the typing of n in the commitment of the conclusion. Note that the required resources in the co-effect of the premise may be smaller than the *bnd* of the contract; for example, n may already have received the time slices *prov*. Furthermore, the number of resources a process P requires changes if it becomes enclosed in an ambient n; i.e., we move to the resource contract T of n, provided the process P satisfies its part of the contract.

The parallel composition rule T-Par makes use of the fairness of the scheduling of time slices in virtually timed ambients. While the branches agree on the required resources *req*, the provided resources and subambients accumulate. It follows from T-Par that several ambients in parallel will at most need as many resources *req* from the parent ambient as the slowest of them. Furthermore, T-Par changes assumptions and commitments depending on the assumptions and the commitments of the composed processes, using the context composition operator from Def. 5 to compose environments. We have $\mathit{dom}(\mathit{\Delta}_P) \cap \mathit{dom}(\mathit{\Delta}_Q) = \varnothing$, which is a consequence of the uniqueness of ambient names. The assumptions of the branches split the resource contracts of the environment $\mathit{\Gamma}$ between the type judgments for P_1 and P_2 and the commitments split such that $\mathit{\Delta}'_1$ is the assumption for P_1 and vice versa. The replication rule T-Rep imposes the restriction that the process being replicated does not incur any cost; allowing that would amount to an unbounded resource need, which cannot be provisioned in a setting with a finite amount of resources.

Now consider the capability rules. In T-Consume, the resource consumption is a requirement to the environment, expressed by increasing the coeffect to $\mathit{req} + 1$. Since the process requires a time slice, it is counted among the currently

$$\textsc{(T-Zero)}\quad \frac{}{\varnothing;\, 0 \vdash \mathbf{0} : \mathtt{ok}\langle 0,0\rangle;\, \varnothing} \qquad \textsc{(T-Tick1)}\quad \frac{}{\varnothing;\, 0 \vdash \mathtt{tick}? : \mathtt{ok}\langle 1,0\rangle;\, \varnothing} \qquad \textsc{(T-Tick2)}\quad \frac{}{\varnothing;\, 0 \vdash \mathtt{tick}! : \mathtt{ok}\langle 1,0\rangle;\, \varnothing}$$

$$\textsc{(T-Ass)}\quad \frac{\Gamma(n) = T}{\Gamma \vdash n : T} \qquad \textsc{(T-Res)}\quad \frac{\Gamma, k : T;\ \mathit{req} \vdash P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle;\ \Delta}{\Gamma;\ \mathit{req} \vdash (\nu k : T)P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle;\ \Delta} \qquad \textsc{(T-Tsub)}\quad \frac{\Gamma \vdash n : T_1 \qquad T_1 \leqslant T_2}{\Gamma \vdash n : T_2}$$

$$\textsc{(T-Amb)}\quad \frac{\begin{array}{c} T' {=} \langle \mathit{cap}, \mathit{bnd}, \mathit{tkn}{+}\mathit{subs}\rangle \quad \mathit{req}{\times}\,\mathit{bnd} \leqslant \mathit{cap}{+}\mathit{prov} \\ \mathit{tkn}{+}\mathit{subs} \leqslant \mathit{bnd} \quad \Gamma';\ \mathit{req} \vdash P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle;\ \Delta \\ \Gamma' = \Gamma, n{:}\langle \mathit{cap}, \mathit{bnd}, \mathit{tkn}\rangle, \mathtt{this}{:}\langle \mathit{cap}, \mathit{bnd}, \mathit{tkn}\rangle \end{array}}{\Gamma;\ \mathit{cap} \vdash \bar{n}[P] : \mathtt{ok}\langle 0, \mathit{bnd}+1\rangle;\ n{:}T', \Delta} \qquad \textsc{(T-Sub)}\quad \frac{\begin{array}{c} \mathit{req}' \leqslant \mathit{req} \quad \mathit{prov} \leqslant \mathit{prov}' \\ \Gamma' \subseteq \Gamma \quad \Delta \subseteq \Delta' \quad \mathit{subs}' \leqslant \mathit{subs} \\ \Gamma';\ \mathit{req}' \vdash P : \mathtt{ok}\langle \mathit{prov}', \mathit{subs}'\rangle;\ \Delta' \end{array}}{\Gamma;\ \mathit{req} \vdash P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle;\ \Delta}$$

$$\textsc{(T-Par)}\quad \frac{\begin{array}{c} \Delta_1 \sim \Delta_1' \qquad \Delta_2 \sim \Delta_2' \qquad \vdash \Gamma : \mathtt{ok} \qquad \vdash \Delta : \mathtt{ok} \\ \Gamma = \Gamma_1 \oplus \Gamma_2 \qquad \Gamma_1 \sim \Gamma_2 \qquad \Gamma_1, \Delta_2';\ \mathit{req} \vdash P_1 : \mathtt{ok}\langle \mathit{prov}_1, \mathit{subs}_1\rangle;\ \Delta_1 \\ \Delta = (\Delta_1 \oplus \Delta_1'), (\Delta_2 \oplus \Delta_2') \qquad \Gamma_2, \Delta_1';\ \mathit{req} \vdash P_2 : \mathtt{ok}\langle \mathit{prov}_2, \mathit{subs}_2\rangle;\ \Delta_2 \end{array}}{\Gamma;\ \mathit{req} \vdash P_1 \mid P_2 : \mathtt{ok}\langle \mathit{prov}_1 + \mathit{prov}_2, \mathit{subs}_1 + \mathit{subs}_2\rangle;\ \Delta}$$

$$\textsc{(T-Consume1)}\quad \frac{\begin{array}{c} \mathit{subs}' = \max\{\mathit{subs}, 1\} \\ \Gamma; \mathit{req} \vdash P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle, \Delta \end{array}}{\Gamma; \mathit{req} + 1 \vdash \mathtt{c}.P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}'\rangle, \Delta} \qquad \textsc{(T-Consume2)}\quad \frac{\begin{array}{c} \mathit{subs}' = \max\{\mathit{subs}, 1\} \\ \Gamma; \mathit{req} \vdash P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle, \Delta \end{array}}{\Gamma; \mathit{req} + 1 \vdash \mathtt{tick}?.P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}'\rangle, \Delta}$$

$$\textsc{(T-In)}\quad \frac{\begin{array}{c} T = \langle \mathit{cap}, \mathit{bnd}, \mathit{tkn}\rangle \qquad T' = \langle \mathit{cap}, \mathit{bnd}, \mathit{tkn} + \mathit{bnd}' + 1\rangle \\ \Gamma, m{:}T; \mathit{req} \vdash P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle, \Delta \qquad \mathit{bnd} \times \mathit{req} \leqslant \mathit{cap} \\ \Gamma \vdash \mathtt{this} : \langle \mathit{cap}', \mathit{bnd}', \mathit{tkn}'\rangle \qquad \mathit{tkn} + \mathit{bnd}' + 1 \leqslant \mathit{bnd} \end{array}}{\Gamma, m{:}T';\ \mathit{req} \vdash \mathtt{in}\, m.P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle;\ \Delta} \qquad \textsc{(T-Rep)}\quad \frac{\begin{array}{c} \Gamma; 0 \vdash P : \mathtt{ok}\langle 0,0\rangle, \Delta_P \\ C \in \{\mathtt{in}\, n, \mathtt{out}\, n, \mathtt{open}\, n\} \end{array}}{\Gamma; 0 \vdash !C.P : \mathtt{ok}\langle 0,0\rangle, \Delta_P}$$

$$\textsc{(T-Out)}\quad \frac{\Gamma; \mathit{req} \vdash P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle, \Delta}{\Gamma; \mathit{req} \vdash \mathtt{out}\, m.P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle, \Delta} \qquad \textsc{(T-Open)}\quad \frac{\Gamma; \mathit{req} \vdash P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle, \Delta}{\Gamma; \mathit{req} \vdash \mathtt{open}\, m.P : \mathtt{ok}\langle \mathit{prov}, \mathit{subs}\rangle, \Delta}$$

Table 3. Type rules for the virtually timed ambients.

hosted processes. If it was already counted as a timed process, *subs* remains unchanged, but since it could have been untimed, we let $\mathit{subs}' = \max\{\mathit{subs}, 1\}$.

Rule T-In derives an assumption about ambient m under which the movement $\mathtt{in}\, m.P$ can be typed. Since the movement involves all processes co-located with $\mathtt{in}\, m.P$, the rule depends on the resource contract of **this**, the ambient in which the current process is located. The rule has a premise expressing that if P can be typed with a resource contract T for m, then $\mathtt{in}\, m.P$ can be typed with the resource contract T' for m. In addition, the hosting capacity bnd' of $\texttt{this}$ and $\texttt{this}$ itself are added to the assumed currently hosted processes tkn of the premise. The premise $bnd \times req \leqslant cap$ expresses that the required resources req must be within the resource capacity cap if scheduled to all processes within the hosting capacity bnd of m. The effect and co-effect carry over directly from the premise, as the movement does not modify the required or provided resources or subambients of P. In contrast, rules T-Open and T-Out simply preserve the co-effect and effect of its premise, since the actual movement is captured by the worst-case assumption in T-Amb.

Example 2 (Typing of in-capabilities). We revisit Example 1 to illustrate the typing of $cloud[\mathbf{0}] \mid vm[\mathtt{in}\, cloud.\mathtt{c}.\mathbf{0}]$. From T-Zero and T-Consume1, we get $\varnothing; 1 \vdash \mathtt{c}.\mathbf{0} : \mathbf{ok}\langle 0,1\rangle; \varnothing$. The $\mathtt{in}$-capability will move the ambient containing this process, which is captured by $\texttt{this}$ in the typing environment. Let us type $\texttt{this}$ by $T = \langle 1,1,1\rangle$. In this case $cloud$ will need a hosting capacity of at least 2, so let us type $cloud$ by $T' = \langle 2,2,2\rangle$. Then, from T-In, we get

$$cloud : T', \texttt{this} : T; 1 \vdash \mathtt{in}\; cloud.\mathtt{c}.\mathbf{0} : \mathbf{ok}\langle 0,1\rangle; \varnothing.$$

By T-Amb, we get $cloud : T'; 1 \vdash vm[\mathtt{in}\;\; cloud.\mathtt{c}.\mathbf{0}] : \mathbf{ok}\langle 0,2\rangle; vm : T$. Similarly, $\varnothing; 2 \vdash cloud[\mathbf{0}] : \mathbf{ok}\langle 0,1\rangle; cloud : \langle 2,2,0\rangle$ and T-Par gives us

$$\varnothing; 2 \vdash cloud[\mathbf{0}] \mid vm[\mathtt{in}\; cloud.\mathtt{c}.\mathbf{0}] : \mathbf{ok}\langle 0,3\rangle; vm : T, cloud : T';$$

Example 3 (Typing of open-capabilities). We consider the typing of a process $cloud[\mathtt{open}\;\; vm.\mathbf{0} \mid vm[\mathtt{c}.\mathbf{0}]]$. From T-Zero and T-Consume, we get $\varnothing; 1 \vdash \mathtt{c}.\mathbf{0} : \mathbf{ok}\langle 0,1\rangle; \varnothing$. Let vm have type $T = \langle 1,1,1\rangle$. Then, by T-Amb,

$$\varnothing; 1 \vdash vm[\mathtt{c}.\mathbf{0}] : \mathbf{ok}\langle 0,2\rangle; vm : T.$$

By T-Zero, T-Open and T-Sub, we have $\varnothing; 1 \vdash \mathtt{open}\;\; vm.\mathbf{0} : \mathbf{ok}\langle 0,0\rangle; \varnothing$. By T-Par, we obtain $\varnothing; 1 \vdash \mathtt{open}\;\; vm.\mathbf{0} \mid vm[\mathtt{c}.\mathbf{0}] : \mathbf{ok}\langle 0,2\rangle; vm : T$. Let $cloud$ have type $T' = \langle 2,2,2\rangle$. By T-Amb, we get

$$\varnothing; 2 \vdash cloud[\mathtt{open}\;\; vm.\mathbf{0} \mid vm[\mathtt{c}.\mathbf{0}]] : \mathbf{ok}\langle 0,3\rangle; vm : T, cloud : T'.$$

Example 4 (Typing of out-capabilities). We consider the typing of a process

$$cloud[vm[\mathtt{out}\; cloud.\mathtt{c}.\mathbf{0}] \mid \mathbf{0}]$$

By T-Zero and T-Consume, we have $\varnothing; 1 \vdash \mathtt{c}.\mathbf{0} : \mathbf{ok}\langle 0,1\rangle; \varnothing$, and by T-Out we get

$$\varnothing; 1 \vdash \mathtt{out}\; cloud.\mathtt{c}.\mathbf{0} : \mathbf{ok}\langle 0,1\rangle; \varnothing$$

Let $T = \langle 1,1,1\rangle$. We can type vm by

$$\varnothing; 1 \vdash vm[\mathtt{out}\; cloud.\mathtt{c}.\mathbf{0}] : \mathbf{ok}\langle 0,2\rangle; vm : T$$

and, with $T' = \langle 2,2,2\rangle$, we get

$$\varnothing; 2 \vdash cloud[vm[\mathtt{out}\; cloud.\mathtt{c}.\mathbf{0}] \mid \mathbf{0}] : \mathbf{ok}\langle 0,3\rangle; vm : T, cloud : T'$$

Example 5 (Failure of type checking). Type checking fails if the provisioning of resources for an incoming ambient in a timely way cannot be statically guaranteed. This can occur for different reasons. One reason is that an ambient may lack *sufficient hosting capacity* to take in the processes that want to enter. Let $T' = \langle 2, 2, 2\rangle$ as before and consider again the process $\mathit{cloud}[\mathbf{0}] \mid \mathit{vm}[\mathtt{in}\ \mathit{cloud}.\mathtt{c}.\mathbf{0}]$ from Example 2. Now assume a second virtual machine $\mathit{vm}_2[\mathtt{in}\,\mathit{cloud}.\mathtt{c}.\mathbf{0}]$ which aims to enter the *cloud* ambient, resulting in the parallel process

$$\mathit{cloud}[\mathbf{0}] \mid \mathit{vm}[\mathtt{in}\ \mathit{cloud}.\mathtt{c}.\mathbf{0}] \mid \mathit{vm}_2[\mathtt{in}\ \mathit{cloud}.\mathtt{c}.\mathbf{0}]$$

We can type vm_2 similarly to vm in Example 2.:

$$\mathit{cloud} : T'; 1 \vdash \mathit{vm}_2[\mathtt{in}\ \mathit{cloud}.\mathtt{c}.\mathbf{0}] : \mathbf{ok}\langle 0, 2\rangle; \mathit{vm}_2 : T.$$

In contrast to Example 2, the hosting capacity for *cloud* in T' cannot accommodate both vm and vm_2; type checking fails when giving *cloud* the resource contract T'. (Remark that type checking would succeed if *cloud* get more resources, e.g., the resource contract $\langle 4, 4, 4\rangle$.)

Another reason is that the resource contract of *cloud* may have a *too low resource capacity.* Consider a third virtual machine $\mathit{vm}_3[\mathtt{in}\ \mathit{cloud}.\mathtt{c}.\mathtt{c}.\mathtt{c}.\mathbf{0}]$ which can be typed with the resource contract $\langle 3, 1, 1\rangle$ for vm_3. Again, type checking $\mathit{cloud}[\mathbf{0}] \mid \mathit{vm}_3[\mathtt{in}\ \mathit{cloud}.\mathtt{c}.\mathtt{c}.\mathtt{c}.\mathbf{0}]$ fails if *cloud* were given the resource contract T', since the resource capacity of *cloud* must here be at least 6 with hosting capacity 2. (Here, *cloud* would need a resource contract such as $\langle 6, 2, 2\rangle$ for the expression to be well-typed.)

Example 6 (Capacity of an ambient). Assume that the process

$$n_1[\mathtt{in}\ m.P_1] \mid n_2[\mathtt{in}\ m.P_2] \mid m[Q]$$

is well-typed. Let $T_1 = \langle \mathit{cap}, \mathit{bnd}, \mathit{tkn}_1\rangle$, $T_2 = \langle \mathit{cap}, \mathit{bnd}, \mathit{tkn}_2\rangle$ and $T_3 = \langle \mathit{cap}, \mathit{bnd}, \mathit{tkn}_3\rangle$ be resource contracts such that

$$m : T_i; \mathit{req}_i \vdash n_i[\mathtt{in}\, m.P_i] : \mathbf{ok}\langle \mathit{prov}_i, \mathit{subs}_i\rangle; \Delta_i$$

for $i \in \{1, 2\}$, and $\varnothing; \mathit{req}_3 \vdash m[Q] : \mathbf{ok}\langle \mathit{prov}_3, \mathit{subs}_3\rangle; m : T_3$. Let $r_{12} = \max(r_1, r_2)$ and $T_{12} = \langle \mathit{cap}, \mathit{bnd}, \mathit{tkn}_1 \oplus \mathit{tkn}_2\rangle$. Since $n_1[\mathtt{in}\ m.P_1] \mid n_2[\mathtt{in}\ m.P_2]$ is well-typed, we have $\mathit{tkn}_1 \oplus \mathit{tkn}_2 \leqslant \mathit{bnd}$ and, by T-Par,

$$m : T_{12}; \mathit{req}_{12} \vdash n_1[\mathtt{in}\ m.P_1] \mid n_2[\mathtt{in}\ m.P_2] : \mathbf{ok}\langle \mathit{prov}_{12}, \mathit{subs}_{12}\rangle; \Delta_{12}$$

where $\mathit{prov}_{12} = \mathit{prov}_1 + \mathit{prov}_2$, $\mathit{subs}_{12} = \mathit{subs}_1 + \mathit{subs}_2$ and $\Delta = \Delta_1, \Delta_2$. By applying T-Par again, we get

$$\varnothing; \mathit{req} \vdash n_1[\mathtt{in}\ m.P_1] \mid n_2[\mathtt{in}\ m.P_2] \mid m[Q] : \mathbf{ok}\langle \mathit{prov}, \mathit{subs}\rangle; m : T, \Delta$$

where $\mathit{req} = \max\{\mathit{req}_{12}, \mathit{req}_3\}$, $\mathit{prov} = \mathit{prov}_{12} + \mathit{prov}_3$, $\mathit{subs} = \mathit{subs}_{12} + \mathit{subs}_3$ and $T = \langle \mathit{cap}, \mathit{bnd}, \mathit{tkn}_{12} + \mathit{tkn}_3\rangle$. Thus, the weakest resource contract which types m and allows both n_1 and n_2 to enter, will have $\mathit{bnd} = \mathit{tkn}_{12} + \mathit{tkn}_3$ and $\mathit{cap} = \mathit{bnd} \times \mathit{req}$.

4 Soundness of Resource Management

The soundness of resource management can be perceived similarly to that of message exchange [14]. We prove a subject reduction theorem, stating that the number of resources required for a boxed process to make progress is preserved under reduction.

Theorem 1 (Subject Reduction). *Assume* $\Gamma, req \vdash n[P] : \textbf{ok}\langle prov, subs\rangle; \Delta$ *and* $n[P] \twoheadrightarrow n[Q]$*, then there are environments* $\Gamma' \leqslant \Gamma$ *and* $\Delta' \leqslant \Delta$ *such that* $\Gamma', req' \vdash n[Q] : \textbf{ok}\langle prov', subs'\rangle; \Delta'$ *and* $req' \leqslant req$ *or* $req' = req \wedge prov' \geqslant prov$*.*

Proof. By induction on the derivation of $n[P] \twoheadrightarrow n[Q]$ (For details, see App. **??**). □

Further, we prove a progress theorem, which shows that a well-typed boxed process which receives the approximated number of resources from its environment will not get stuck due to missing resources. Obviously, a well-typed process may be non-progressing due to other reasons. For instance, the terminated process $\mathbf{0}$ cannot "proceed" no matter how many ticks it may be served. To characterize a situation where inside the process, there is a sub-process in need of a tick to proceed, be it an unserved ambient or a process guarded by a $\texttt{tick}?$-capability, we use the contextual variant of barbs from Def. 2 .

Theorem 2 (Tick progress). *Assume* $\Gamma; req \vdash P : \textbf{ok}\langle prov, subs\rangle; \Delta$ *where* $P = m[R]$*, and let* $Q = \bar{n}[P \mid \texttt{tick}! \mid \ldots \mid \texttt{tick}!]$ *, where* P *is running in parallel with req occurrences of* $\texttt{tick}!$ *inside some enclosing ambient. If* $Q{\downarrow}^{\mathcal{C}}_{\texttt{tick}?}$ *for some context* $\mathcal{C}$*, then* $Q \twoheadrightarrow Q'$ *for some process* Q'*.*

Proof sketch. This follows from the definition of the typing rules. If P contains the subprocess $\texttt{c}.P'$ it follows from the typing rule for the consume capability that $req \geqslant 1$. From the other typing rules it the number of resources is sufficient to trigger the reduction $\texttt{c}.P' \twoheadrightarrow P'$. Thus, Q can reduce to Q'. □

With the properties of subject reduction and progress the type system guarantees the soundness of resource management.

Corollary 1 (Soundness). *The type system guarantees the soundness of resource management, i.e., the transitive closure of the progress result holds.*

5 Related Work

We first discuss related work on modeling virtualization, time and resources, mainly focusing on process algebra, and then related work on type systems.

The calculus presented here differs from Stumpf *et al*'s original work on virtually timed ambients [4,9] by assuming uniform time and by the use of freezing and unfreezing operations, which allow a significantly simpler formulation of the calculus. The behavior of the original calculus, with non-uniform time, can be recaptured by modifying the rule **tick**? $\rightharpoondown$ **tick**! to cater for different numbers of input and output ticks, and to contextualize the rule for specific ambients. Stumpf *et al.* provide more elaborate examples of how aspects of virtualization (such as scaling and load balancing) can be modeled in virtually timed ambients (e.g., [4, 9]). For the original calculus of virtually timed ambients, a modal logic with somewhere and sometime modalities [15] captures aspects of reachability for these ambients. Whereas this work can express more complex properties of a given process than the contract-based types in our paper, the logic cannot capture properties for all processes, in contrast to our work.

Gordon proposed a simple formalism for virtualization loosely based on mobile ambients [16]. Virtually timed ambients [4] stay closer to the syntax of the original ambient calculus, while including notions of time and resources. This model of resources as processing capacity over time builds on deployment components [17, 18], a modeling abstraction for cloud computing in ABS [19]. Compared to virtually timed ambients, ABS does not support nested deployment components nor the timed capabilities of ambients.

Timers have been studied both for the distributed π-calculus [20, 21] and mobile ambients (e.g., [22]) to express the possibility of a timeout, controlled by a global clock. In membrane computing, rule execution similarly takes exactly one time unit, as given by a global clock [23]. Timed P systems [24] overcome this restriction by associating with each rule an integer representing the time needed to complete its execution. This resembles the timer approach on mobile ambients [22]. In contrast, schedulers in virtually timed ambients recursively control the execution power of the nested location structure. Modeling timeouts is a straightforward extension of virtually timed ambients.

A process algebra with resources as primitives has been proposed in ACSR [25]. In contrast to the **c**-capability in virtually timed ambients, ACSR uses a set of consume actions with a priority relation, which can be used to encode, e.g., scheduling policies. PADS [26] extends ACSR with hierarchical approaches to scheduling, making the provisioning of resources explicit and introducing refinement relations on supply and demand. PARS [27] similarly uses explicit resource provisioning to specify that process needs, e.g., one processor and 100 units of memory for a given duration. Neither of these calculi combine resources with locations and mobility. The Kell calculus [28] supports mobility, inspired by mobile ambients, through higher order communication, but does not model resource provisioning. Whereas Kell has a type system to enforce the uniqueness of names [28], none of these calculi provide contract-based abstractions for resource analyses such as our type system for resource contracts.

A type system for the ambient calculus was defined in [14] to control communication and mobility. For communication, a basic ambient type captures the kind of messages that can be exchanged within. For mobility, the type system controls which ambients can enter. Types are often enriched with effects to capture the aspects of computation which are not purely functional. In process algebra, session types have been used to capture communication in the π-calculus [29]. Orchard and Yoshida have shown that effects and session types are similar concepts as they can be expressed in terms of each other [30]. Session types have been defined for boxed ambients in [31] and behavioral effects for the ambient calculus in [32], where the original communication types by Cardelli and Gordon are enhanced by movement behavior. This is captured with traces, the flow-sensitivity hereby results from the copying of the capabilities in the type. Type-based resource control for resources in the form of locks has been proposed for process algebras in general [33] and for the π-calculus in particular [34, 35].

The idea of assumptions and commitments (or relies and guarantees) is quite old, but has mainly been explored for specification and compositional reasoning about concurrent or parallel processes (e.g., [36–38]). Assumption commitment style type systems have previously been used for multi-threaded concurrency [39, 40]; the resources controlled by the effect-type system there are locks and a general form of futures, in contrast to our work.

To capture how a computation depends on an environment instead of how the computation affects it, Petricek, Orchard and Mycroft suggest the term *coeffect* as a notion of *context-dependent* computation [7,8]. Dual to effects, which can be modeled monadically, the semantics of coeffects is provided by indexed comonads [41, 42]. We use coeffects to control time and resources. An approach to control timing via types can be found in [43], which develops types and typed timers for the timed π-calculus. Another approach to resource control without coeffects can be found in [44], which proposes a type system to restrict resource access for the distributed π-calculus. In [45] a type system for resource control for a fragment of the mobile ambients is defined by adding capacity and weight to communication types for controlled ambients. Simplified non-modifiable mobile ambients with resources, and types to control migration and resource distribution are proposed in [46]. Another fragment of the ambient calculus, finite control ambients with only finite parallel composition, are covered in [47]. Here the types are a bound to the number of allowed active outputs in an ambient.

6 Concluding Remarks

Virtualization opens for new and interesting models of computation by explicitly emphasizing deployment and resource management. This paper introduces a type system based on resource contracts for virtually timed ambients, a calculus of hierarchical locations of execution with explicit resource provisioning. Resource provisioning in this calculus is based on virtual time, a local notion of time reminiscent of time slices provisioned by operating systems in the context of nested virtualization. The proposed assumption-commitment type system with effects and coeffects enables static checking of timing and resource constraints for ambients and gives an upper bound on the resources used by a process. The type system supports subsumption, which allows relating subtypes to supertypes. We show that the proposed type system is sound in terms of subject reduction and a progress property. Although these are core properties for type systems, the results are here given for a non-standard assumption-commitment setting in an operational framework. The type system further provides reusable properties as it supports abstraction and the results would also hold for other operational accounts of fair resource distribution. The challenge of how to further generalize the distribution strategy and type system for, e.g., earliest deadline first or priority-based scheduling policies, remains.

The virtually timed ambients used for the models in this paper extend the basic ambient calculus without channel communication. Introducing channels would lead to additional synchronization, which could potentially be exploited to derive more precise estimations about resource consumption. Such an extension would be non-trivial as the analysis of the communication structure would interfere with scheduling.

References

1. Goldberg, R.P.: Survey of virtual machine research. IEEE Computer **7**(6) (1974) 34–45
2. Ben-Yehuda, M., Day, M.D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., Yassour, B.: The Turtles project: Design and implementation of nested virtualization. In: Proceedings 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010), USENIX Association (2010) 423–436
3. Williams, D., Jamjoom, H., Weatherspoon, H.: The Xen-Blanket: Virtualize once, run everywhere. In: Proceedings 7th European Conference on Computer Systems (EuroSys'12), ACM (2012) 113–126
4. Johnsen, E.B., Steffen, M., Stumpf, J.B.: A calculus of virtually timed ambients. In James, P., Roggenbach, M., eds.: Postproceedings of the 23rd International Workshop on Algebraic Development Techniques (WADT 2016). Volume 10644 of Lecture Notes in Computer Science., Springer (2017) 88–103
5. Cardelli, L., Gordon, A.D.: Mobile ambients. Theoretical Computer Science **240**(1) (2000) 177–213
6. Giovannetti, E.: Ambient calculi with types: a tutorial. In: Global Computing — Programming Environments, Languages, Security and Analysis of Systems. Volume 2874 of Lecture Notes in Computer Science., Springer (2003) 151–191
7. Petricek, T., Orchard, D., Mycroft, A.: Coeffects: A calculus of context-dependent computation. In Jeuring, J., Chakravarty, M.M.T., eds.: Proceedings of the International Conference on Functional Programming (ICFP'14), ACM (2014)
8. Petricek, T., Orchard, D., Mycroft, A.: Coeffects: unified static analysis of context-dependence. In Fomin, F.V., Freivalds, R., Kwiatkowska, M.Z., Peleg, D., eds.: Proceedings of the International Conference on Automata, Languages, and Programming (ICALP'13). Volume 7966 of Lecture Notes in Computer Science., Springer (2013) 385–397
9. Johnsen, E.B., Steffen, M., Stumpf, J.B.: Virtually timed ambients: A calculus of nested virtualization. Journal of Logical and Algebraic Methods in Programming **94** (2018) 109 – 127
10. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java bytecode. In De Nicola, R., ed.: Proc. ESOP. Volume 4421 of Lecture Notes in Computer Science., Springer (2007) 157–172
11. Albert, E., Correas, J., Johnsen, E.B., Pun, V.K.I., Román-Díez, G.: Parallel cost analysis. ACM Trans. Comput. Log. **19**(4) (2018) 31:1–31:37

12. Milner, R., Sangiorgi, D.: Barbed bisimulation. [48] 685–695
13. Merro, M., Zappa Nardelli, F.: Behavioral theory for mobile ambients. Journal of the ACM **52**(6) (2005) 961–1023
14. Cardelli, L., Ghelli, G., Gordon, A.D.: Types for the ambient calculus. Information and Computation **177**(2) (2002) 160–194
15. Johnsen, E.B., Steffen, M., Stumpf, J.B., Tveito, L.: Checking modal contracts for virtually timed ambients. In Fischer, B., Uustalu, T., eds.: Proc. 15th Intl. Colloquium on Theoretical Aspects of Computing (ICTAC 2018). Volume 11187 of Lecture Notes in Computer Science., Springer (2018) 252–272
16. Gordon, A.D.: V for virtual. Electronic Notes in Theoretical Computer Science **162** (2006) 177–181
17. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. Journal of Logic and Algebraic Methods in Programming **84**(1) (2015) 67–91
18. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: An industrial case study using Real-Time ABS. Journal of Service-Oriented Computing and Applications **8**(4) (2014) 323–339
19. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In Aichernig, B., de Boer, F.S., Bonsangue, M.M., eds.: Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Volume 6957 of Lecture Notes in Computer Science., Springer (2011) 142–164
20. Berger, M.: Towards Abstractions for Distributed Systems. PhD thesis, University of London, Imperial College (2004)
21. Prisacariu, C.: Timed distributed pi-calculus. In: Modelling and Verifying of Parallel Processes (MOVEP06). (2006) 348–354
22. Aman, B., Ciobanu, G.: Mobile ambients with timers and types. In Jones, C.B., Liu, Z., Woodcock, J., eds.: Proceedings 4th International Colloquium on Theoretical Aspects of Computing (ICTAC'07). Volume 4711 of Lecture Notes in Computer Science., Springer (2007) 50–63
23. Paun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
24. Cavaliere, M., Sburlan, D.: Time-independent P systems. In Mauri, G., Paun, G., Pérez-Jiménez, M.J., Rozenberg, G., Salomaa, A., eds.: Proceedings 5th International Workshop on Membrane Computing (WMC'04). Volume 3365 of Lecture Notes in Computer Science., Springer (2004) 239–258
25. Lee, I., Philippou, A., Sokolsky, O.: Resources in process algebra. Journal of Logic and Algebraic Programming **72**(1) (2007) 98 –122
26. Philippou, A., Lee, I., Sokolsky, O.: PADS: an approach to modeling resource demand and supply for the formal analysis of hierarchical scheduling. Theor. Comput. Sci. **413**(1) (2012) 2–20
27. Mousavi, M.R., Reniers, M.A., Basten, T., Chaudron, M.R.V.: PARS: A process algebraic approach to resources and schedulers. In Alexander, M., Gardner, W., eds.: Process Algebra for Parallel and Distributed Processing. Chapman and Hall/CRC (2008)
28. Bidinger, P., Stefani, J.: The Kell calculus: Operational semantics and type system. In Najm, E., Nestmann, U., Stevens, P., eds.: Proceedings 6th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003). Volume 2884 of Lecture Notes in Computer Science., Springer (2003) 109–123

29. Honda, K.: Types for dyadic interaction. [49] 509–523
30. Orchard, D., Yoshida, N.: Effects as sessions, sessions as effects. In: POPL 2016, ACM Press (2016)
31. Garralda, P., Compagnoni, A., Dezani-Ciancaglini, M.: BASS: Boxed Ambients with Safe Sessions. In Maher, M., ed.: PPDP'06, ACM Press (2006) 61–72
32. Amtoft, T.: Flow-sensitive type systems and the ambient calculus. Higher-Order and Symbolic Computation **21**(4) (2008) 411–442
33. Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Trans. Program. Lang. Syst. **27**(2) (2005) 264–313
34. Kobayashi, N., Suenaga, K., Wischik, L.: Resource usage analysis for the π-calculus. Logical Methods in Computer Science **2**(3) (2006)
35. Kobayashi, N., Sangiorgi, D.: A hybrid type system for lock-freedom of mobile processes. ACM Trans. Program. Lang. Syst. **32**(5) (2010) 16:1–16:49
36. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems **17**(3) (1995) 507–534
37. Jones, C.B.: Tentative steps towards a development method for interfering programs. ACM Transactions on Programming Languages and Systems **5**(4) (1983) 596–619
38. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Transactions on Software Engineering **7** (1981) 417–426
39. Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. Journal of Logic and Algebraic Programming **78**(7) (2009) 491–518 (28 pages) Special issue with selected contributions of NWPT'07. The paper is a reworked version of an earlier UiO Technical Report TR-364, Oct. 2007.
40. Ábrahám, E., Grüner, A., Steffen, M.: Dynamic heap-abstraction for open, object-oriented systems with thread classes (extended abstract). [50] 1–10 (10 pages) A preliminary version has been included in the informal workshop proceedings of Cosmicah'05, as Queen Mary Technical Report RR-05-04, a longer version has been published as Technical Report 0601 of the Institute of Computer Science of the University Kiel, January 2006.
41. Katsumata, S.: Parametric effect monads and semantics of effect systems. [51] 633–645
42. Uustalu, T., Vene, V.: Comonadic notions of computation. Electronic Notes in Theoretical Computer Science **203** (2008) 263–284 Proceedings 9th Intl. Workshop on Coalgebraic Methods in Computer Science (CMCS 2008).
43. Berger, M., Yoshida, N.: Timed, distributed, probabilistic, typed processes. In: Asian Symposium on Programming Languages and Systems, Springer (2007) 158–174
44. Hennessy, M., Riely, J.: Resource access control in systems of mobile agents. Information and Computation **173**(1) (2002) 82 – 120
45. Teller, D., Zimmer, P., Hirschkoff, D.: Using ambients to control resources. In: Proceedings of the 13th International Conference on Concurrency Theory. CONCUR '02, London, UK, Springer (2002) 288–303
46. Godskesen, J.C., Hildebrandt, T., Sassone, V.: A calculus of mobile resources. In Brim, L., Křetínský, M., Kučera, A., Jančar, P., eds.: Proceedings 13th International Conference on Concurrency Theory (CONCUR 2002). Volume 2421 of Lecture Notes in Computer Science., Springer (2002) 272–287
47. Charatonik, W., Gordon, A.D., Talbot, J.M.: Finite-control mobile ambients. In: European Symposium on Programming. Volume 2305 of LNCS., Springer (2002) 295–313

48. Kuich, W., ed.: Nineteenth Colloquium on Automata, Languages and Programming (ICALP) (Wien, Austria). In Kuich, W., ed.: Proceedings of ICALP '92. Volume 623 of Lecture Notes in Computer Science., Springer (1992)
49. Best, E., ed.: CONCUR '93: Fourth International Conference on Concurrency Theory (Hildesheim, Germany). In Best, E., ed.: Proceedings of CONCUR '93. Volume 715 of Lecture Notes in Computer Science., Springer (1993)
50. Beckmann, A., Berger, U., Löwe, B., Tucker, J.V., eds.: Proceedings of Logical Approaches to Computational Barriers: Second Conference on Computability in Europe, CiE 2006, Swansea, UK, June 30-July 5, 2006. In Beckmann, A., Berger, U., Löwe, B., Tucker, J.V., eds.: Logical Approaches to Computational Barriers: CiE 2006. Volume 3988 of Lecture Notes in Computer Science., Springer (July 2006)
51. ACM: 43th Symposium on Principles of Programming Languages (POPL). In: Proceedings of POPL '14, ACM (2014)