Technical Report

SADRA: Sound Capability-based Access Control System for Resource-Disaggregated Architectures

Contents

1 Introduction **5**

2 Background **9**
2.1 Capability–Object Model 9
2.2 Assumption-Commitment Method 9
2.3 Threat Model 11

3 Related Work **12**
3.1 Criteria 12
3.2 Results 12
3.3 State-of-the-art and Comparison 13

4 Proposed Access-Control System **14**
4.1 Abstract Model of a Disaggregated System 14
4.2 Distributed Access Controllers 14
4.3 Capability Model 15
4.3.1 Indicator Flag 15
4.3.2 Capability Tree 16
4.3.3 Capability Operations 16
4.4 Formal Definitions 18

5 System Design's Soundness **21**
5.1 Well-Formed Capability Trees 21
5.2 Valid Capability 24
5.3 Security Properties 25
5.4 First Lemma: Resource Allocation 25
5.4.1 Resource Allocation - Intermediate Controller 26
5.4.2 Resource Allocation - Resource Controller 28
5.4.3 Resource Allocation - Parallel Composition 30
5.5 Second Lemma: Internal Delegation 30
5.5.1 internal Delegation - Intermediate Controller 31
5.6 Third Lemma: External Delegation 34
5.6.1 External Delegation - Delegating-Side Intermediate Controller 34
5.6.2 External Delegation - Resource Controller 37
5.6.3 External Delegation - Receiving-Side Intermediate Controller 39
5.6.4 External Delegation - Parallel Composition 41
5.7 Fourth Lemma: External Revocation 43
5.7.1 External Revocation - Intermediate Controller 44
5.7.2 External Revocation - Resource Controller 47
5.7.3 External Revocation - Parallel Composition 49
5.8 Fifth Lemma: Internal Revocation 50
5.8.1 Internal Revocation - Parallel Composition 54
5.8.2 Capability Safety 55

5.8.3 Authority Safety 57
5.9 Isolation Property 58

6 Use Case **61**
6.1 Resource Node 61
6.1.1 Resource Controller 62
6.2 PE Node 66
6.2.1 Operating System 66
6.2.2 Intermediate Controller 69
6.3 Parallel Composition 81
6.4 Proof 82
6.4.1 First Lemma: Resource Allocation 82
6.4.2 Second Lemma: Internal Delegation 87
6.4.3 Third Lemma: External Delegation 90
6.4.4 Forth Lemma: Internal Revocation 97
6.4.5 Fifth Lemma: External Revocation 100

7 Model Checking **105**

List of Tables

1 Capability-based Access-control Systems 10
2 Function Definitions 24
3 Correctness Checks of the Models 105

1 Introduction

Modern cloud, datacenter, and high-performance computing architectures increasingly adopt resource disaggregation, decoupling compute, memory, storage, and accelerators into independently managed resource pools connected via high-speed fabrics [31, 4, 12, 35, 44, 52]. This architecture enables dynamic scaling, improved utilization, and flexible workload placement across heterogeneous systems. However, it also introduces a critical challenge: disaggregated systems break traditional access control assumptions. Processes now issue direct read and write requests to remote resources over the network—often bypassing local enforcement and exposing gaps in multi-tenant and decentralized access control—gaps SADRA directly addresses.

Driven by the need for elasticity and resource efficiency, resource disaggregation has gained widespread adoption across industry and academia [39, 21, 3, 22, 24, 34, 8, 23, 2, 14]. It now forms the foundation of modern *data centers*[30, 50, 40], *blade servers*[38], *rack-scale systems*[59, 47, 51], *cloud frameworks*[36, 7, 10], and *high-performance computing* [63, 33].

Resource disaggregation fragments the trust boundary and exposes new attack surfaces: resources are now distributed, remotely accessed, and increasingly reachable through unmediated channels. This undermines host-centric access control models that rely on kernel mediation, co-located policy enforcement, and static privilege assumptions. In such architectures, tenant processes issue read/write requests to remote memory or devices over the network—often bypassing trusted OSes. As shown in Figure 1, access is typically mediated by read/write controllers on the resource node—either dedicated servers or hardware-only datapaths.

One-sided communication—especially when implemented via hardware-only controllers—enables direct memory access without mediation by the remote OS or runtime software. This weakens enforcement of confidentiality, integrity, and accountability. Prior work demonstrates concrete risks: unauthorized reads [55], data corruption and privilege escalation [61], and denial-of-service from congestion [64]. Without fine-grained access control along data paths, resource-disaggregated architectures risk subverting core security guarantees in multi-tenant compute environments [62].

An access control system for disaggregated environments must mitigate emerging risks without trusted host software, while supporting elasticity and heterogeneity. It must operate across diverse compute nodes (e.g., CPUs, FPGAs) and memory types (e.g., byte-addressable *Non-Volatile Memory* in *Memory-Driven Computing* [32]), and enforce policies across software-managed and hardware-only paths. These challenges intensify in multi-tenant deployments, where untrusted processes from different users share compute nodes. Without strong compute-side enforcement, malicious tenants may exploit delegation to escalate privileges or evade revocation. Meanwhile, distributed applications span multiple nodes, requiring secure cross-node coordination of authority—without trusted kernels or static configurations.

Existing access control models—especially those for centralized, monolithic systems—often fall short under the combined pressures of multi-tenancy, heterogeneity, and decentralized enforcement, where trust boundaries fragment and policies must scale across nodes. Among modern alternatives, capability-based access control [17,

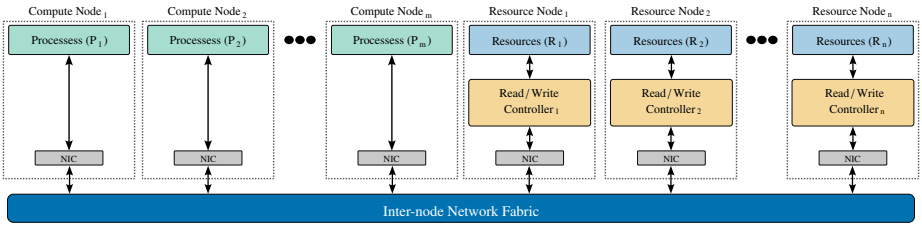


Figure 1: Abstract architecture of a resource-disaggregated system. Compute nodes execute processes that access remote resources via a high-speed interconnect, often without enforcement. Each resource node includes a read/write controller, realized as either a dedicated server or hardware.

20, 11] has emerged as a strong foundation for securing resource-disaggregated architectures [9]. Capability systems enforce least privilege [57] using explicit, unforgeable tokens of authority—called capabilities—that can be validated locally, without global policy state or coordination. This model fits disaggregated environments: it enables fine-grained delegation, decentralized enforcement, and explicit authority boundaries, mitigating confused deputy attacks [53].

As part of our *systematization effort*, we reviewed capability-based access control systems across distributed, OS-level, and hardware-assisted designs [48, 29, 28, 18, 13, 65, 11, 66, 56, 46, 54, 25, 1, 58, 37, 6, 19, 5, 26]. This survey identified six core characteristics essential for secure access control in disaggregated systems: **distributed enforcement**, **subject granularity** (per-process control), **object granularity** (scoped access), **capability delegation**, **capability revocation**, and **capability persistency** across failures.

These characteristics are not merely desirable—they are required to support real-world disaggregated workloads. For instance, in a genomics pipeline deployed on memory-disaggregated infrastructure (Figure 2), distributed tasks must coordinate memory access (distributed enforcement), operate on specific regions (subject/object granularity), delegate and revoke permissions (delegation/revocation), and survive system faults (persistency). Missing any of these undermines correctness or opens security gaps.

Existing systems fail to support all six characteristics simultaneously, limiting their applicability to disaggregated environments. For example, CHERI [65] and L4 [37] assume single-node enforcement; CEP [5] handles memory-side access but spawns a thread per request, exposing it to DoS attacks; SemperOS [26] distributes delegation but requires kernel coordination for revocation—adding latency and fragile failure paths. These are not incidental flaws but stem from fundamental tradeoffs: distributed enforcement complicates revocation, persistency risks stale capabilities, and fine-grained control increases overhead. Extending prior systems often strengthens one axis at the expense of another. A clean-slate design must reconcile all six characteristics across decentralized, multi-tenant, and untrusted infrastructures—balancing granularity, revocation, trust minimization, and elasticity within a fully distributed architecture, without sacrificing performance or deployability.

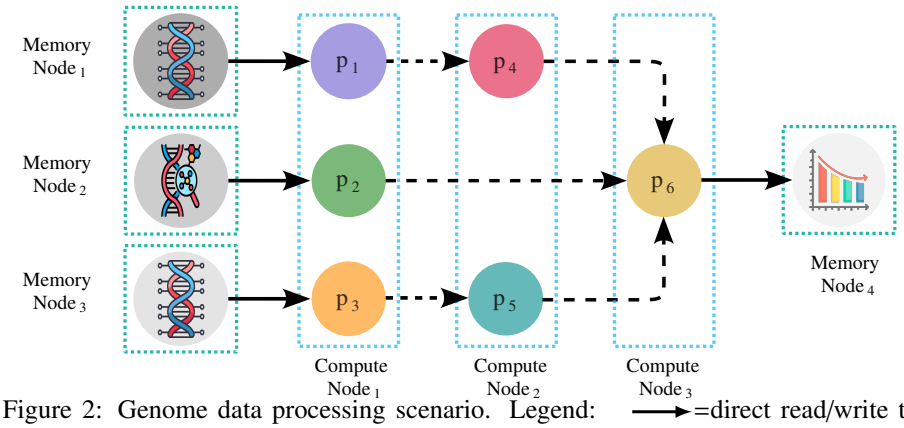


Figure 2: Genome data processing scenario. Legend: ⟶=direct read/write to memory; - - ►=write to memory followed by a read from the same location.

To address these challenges, we present **SADRA**: a sound, general-purpose, and fully distributed capability-based access control system for resource-disaggregated architectures. SADRA supports heterogeneous resources, elastic deployment, and satisfies all six security-relevant characteristics identified in our systematization. Its architecture-agnostic design enables broad applicability—for example, memory-disaggregated systems can restrict access to shared NVM pools, while GPU clusters [42] and Grid platforms [15] can secure distributed compute and accelerator resources. The current implementation targets SmartNIC-enabled resource disaggregation, but SADRA's enforcement model generalizes to other architectures—such as CXL or RDMA—so long as access is mediated at interconnect boundaries

SADRA achieves scalable, low-latency enforcement through a Two-Stage Access Control model offloaded entirely to SmartNICs at both compute and resource nodes. The compute-side SmartNIC performs the first access check and handles local delegation and revocation, enabling fast, tenant-local decisions. The resource-side SmartNIC performs a second, final check and manages remote delegation and revocation, including efficient invalidation of capability hierarchies. By offloading enforcement into hardware, SADRA isolates control from untrusted hosts, reduces coordination overhead, and ensures secure, low-latency access control for high-performance, multi-tenant environments.

We formally prove that SADRA satisfies three core security properties: capability safety, authority safety, and strong isolation (see **??**). Capability safety ensures processes use only valid, legitimately acquired capabilities. Authority safety guarantees delegation does not amplify privilege. Strong isolation ensures a process can allocate a resource and exclude others, enforcing exclusive ownership in shared, disaggregated environments. These proofs are built over our internal capability model using the assumption–commitment (A–C) method (§2.2) and verified via the SPIN model checker [27].

Our analysis shows that capability systems must be fundamentally redesigned for resource-disaggregated environments. The six characteristics are not independent; addressing one can undermine another. Distributed enforcement complicates revocation,

persistency risks stale capabilities, and fine-grained granularity increases overhead. Incremental fixes are inadequate. A new design is needed—one that reconciles all six characteristics while enforcing access securely across heterogeneous, multi-tenant, and untrusted infrastructure. To our knowledge, SADRA is the first system to fully support all six capability-based access control characteristics—distributed enforcement, subject and object granularity, delegation, revocation, and persistency—and to realize them in a practical, hardware-deployed system.

Our contributions are:

- **System architecture.** We design **SADRA**, a sound, general-purpose capability-based access control system for resource-disaggregated architectures. To our knowledge, it is the first system to fully support six core characteristics—distributed enforcement, subject and object granularity, delegation, revocation, and persistency—and demonstrate their integration in a deployed prototype.
- **Enforcement model.** We propose a novel two-stage capability enforcement mechanism offloaded to SmartNICs. It performs local and remote checks, delegation, and revocation in hardware, enabling scalable, kernel-independent access control in multi-tenant settings.
- **Formal guarantees.** We prove that SADRA satisfies capability safety, authority safety, and strong isolation using the assumption–commitment method (see appendix), and verify these properties via SPIN model checking.
- **Instantiation.** We implement SADRA for memory-disaggregated systems and show it enables secure, revocable access to shared memory.
- **Evaluation.** We deploy SADRA on FPGA-based SmartNICs and evaluate it across representative read/write workloads, showing worst-case overhead below 0.36% even during revocation and delegation.

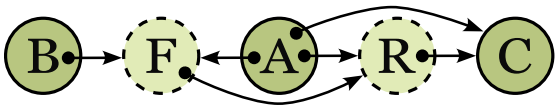


Figure 3: Capability-Object Model. In this model, F is the forwarding proxy and R is the revoking proxy.

2 Background

We begin by outlining the preliminaries before presenting our design.

2.1 Capability–Object Model

Our access control system adopts and extends the classical *object–capability model* [43]—where all entities are *objects* and authority is conferred via unforgeable references called *capabilities* [17, 20]—to operate entirely in SmartNICs for hardware-only, distributed enforcement over resources or principals. A capability designates and authorizes access to a target object, bundling identity and rights. Using a capability is the only way to invoke an object, eliminating ambient authority and enforcing least privilege.

Capabilities are created, distributed, and revoked by the SmartNIC-based enforcement layer, ensuring *capability safety* against untrusted hosts and peers; capabilities are obtained through controlled channels, preventing forgery or replay. This enables a composable, delegatable, and dynamically revocable access control system enforced at both compute and resource nodes, without reliance on host software (§2.3).

Delegation is explicit: a holder hands a capability to another object. To enable revocation, delegation is mediated by *interposed proxies*: if object A holds a capability to C and wishes to delegate to B, A creates R (revoking proxy) and F (forwarding proxy). A grants B a capability to F, which forwards to C via R. Disabling R makes F ineffective (Figure 3). This proxy pattern, adapted to our two-stage SmartNIC enforcement, provides *revocation soundness*: once revoked, subsequent requests are denied before reaching the resource—immediately at the local stage for intra-node revocation, and on first access at the resource side for inter-node revocation, with no interim access while enforcement points update asynchronously.

2.2 Assumption-Commitment Method

To verify SADRA's soundness, we employ the assumption–commitment (A–C) technique [45, 16], a standard *compositional-analysis* method for concurrent and distributed systems. The A–C technique enables modular reasoning: each component is verified in isolation, assuming an environment that respects certain assumptions, and in turn committing to guarantees that the environment can rely on.

Table 1: Capability-based Access-control Systems

	CAP [48]	StarOS [29]	IBM/38 [28]	Hardbound [18]	iAPX432 [13]	CHERI [65]	M-Machine [11]	Hydra [66]	Chorus [56]	Amoeba [46]	Accent [54]	KeyKOS [25]	Mach [1]	EROS [58]	L4 [37]	Barrelfish [6]	CEP [5]	SemperOS [26]	SADRA
Distributed	○	●	○	○	●	○	○	○	●	●	●	○	●	○	○	●	◐	●	●
Subject Granularity	●	◐	●	●	●	●	●	●	●	●	●	●	●	●	●	●	◐	●	●
Object Granularity	●	●	●	◐	●	●	●	●	◐	◐	◐	◐	◐	◐	◐	◐	●	●	●
Capability Delegation	●	●	●	○	○	●	●	●	●	●	◐	●	●	●	●	●	●	●	●
Capability Revocation	○	○	●	○	○	○	●	○	●	◐	◐	○	◐	●	○	●	●	◐	●
Persistency	●	○	○	○	●	○	●	○	◐	●	○	◐	◐	●	●	○	●	◐	●
HW/SW Demands	ISA/OS	ISA/OS	ISA	ISO/C	ISA	ISA/C	HW/ISA	OS	OS	OS	OS	OS	OS	OS	OS	OS	Co-P	HW/OS	Co-P

Legend: ●=fully; ◐=partially; ○=no
C: Compiler; Co-P: Co-Processor; ISA: Instruction Set Architecture; OS: Operating System

2.2.0.1 Programs and Specifications.

The technique models each module as a finite-state machine, referred to as a *program* p. Correctness is expressed by an A–C formula:

$$\langle A,\ C\rangle\colon\ \{\varphi\}\ p\ \{\psi\}$$

where φ is the *precondition* on p's initial state, ψ is the *postcondition* on p's final state, A is the *assumption* about inputs/messages received from the environment, and C is the *commitment* guaranteed by p about its outputs/messages. A program satisfies the specification if it delivers C whenever the environment respects A.

2.2.0.2 Parallel Composition.

The correctness of the overall system is obtained by combining modules using the *Parallel Composition Rule*:

Rule 1. *For programs $p = p_i \parallel p_j$, if*

$$\frac{\begin{array}{c}\langle A_i, C_i\rangle\colon \{\varphi_i\}\ p_i\ \{\psi_i\}\\ \langle A_j, C_j\rangle\colon \{\varphi_j\}\ p_j\ \{\psi_j\}\\ A \wedge C_i \rightarrow A_j \quad A \wedge C_j \rightarrow A_i\end{array}}{\langle A, C_i \wedge C_j\rangle\colon \{\varphi_i \wedge \varphi_j\}\ (p_i \parallel p_j)\ \{\psi_i \wedge \psi_j\}}\ \begin{array}{l}\textit{Parallel}\\ \textit{Composition}\end{array}$$

Here $\langle A, C_i \wedge C_j\rangle$ captures the environment-facing assumptions and commitments of the composite program. The side conditions require that if A_i makes assumptions about channels jointly used with p_j, then C_j must guarantee them (and vice versa). For assumptions about external channels, A must support them. Verifying the implications $A \wedge C_i \rightarrow A_j$ and $A \wedge C_j \rightarrow A_i$ suffices to ensure the global correctness formula holds.

2.2.0.3 Application to SADRA.

In our setting, each SmartNIC controller (compute or resource) is modeled as a program p with assumptions about the messages it receives on its network channels, and commitments about the validity and enforcement of capability operations it emits. The A–C specification ensures that as long as remote peers and local software respect the stated assumptions, the SmartNIC guarantees capability safety and well-formedness properties. By applying the Parallel Composition Rule to all SmartNIC programs and their communication channels, we obtain a system-wide correctness formula, thereby establishing SADRA's soundness.

2.3 Threat Model

We target multi-tenant, resource-disaggregated cloud datacenters with untrusted hosts. Compute nodes issue operations over a high-speed fabric to resource nodes hosting the resource (e.g., memory). Security depends on capability checks at both compute and resource nodes, enforced in hardware by programmable SmartNICs mediating all traffic in a two-stage pipeline. The adversary may control any untrusted component, including its software stack and network communication.

Trusted components are the SmartNIC hardware and firmware running our enforcement logic, the on-NIC capability store (with delegation and revocation metadata), and the enforcement pipeline that validates capabilities before reaching resources. Untrusted components are host OSs, hypervisors, guest VMs or containers, tenant software, and the network fabric unless authenticated and encrypted by the SmartNIC.

Our design guarantees: (1) **Authority safety** – no subject can gain or delegate authority beyond what it holds, enforced by on-NIC checks to prevent over-delegation and cross-node privilege escalation; (2) **Capability integrity** – capabilities cannot be forged, altered, or bypassed, as each request is validated in hardware before reaching the resource; (3) **Least privilege** – subjects are issued only the capabilities needed for their tasks, with others obtained solely through explicit delegation; and (4) **Distributed enforcement consistency** – checks at compute and resource nodes yield equivalent decisions, even under asynchronous revocation, preventing bypass via a compromised node. Other goals, including capability safety and revocation soundness, are in §**??**.

We exclude from the threat model physical compromise of SmartNIC hardware, hardware side-channel attacks (e.g., timing or power analysis), and denial-of-service on the network or SmartNIC, as these do not affect the correctness of our enforcement logic. These assumptions and commitments are formalized in the security proof using the Assumption–Commitment (A–C) method (§2.2).

3 Related Work

Building on the comparative methodology of Bresniker et al. [9], we conducted a targeted review of academic literature [48, 29, 28, 18, 13, 65, 11, 66, 56, 46, 54, 25, 1, 58, 37, 6, 19, 5, 26] covering historical and modern capability systems—both hardware and OS-based—relevant to resource-disaggregated architectures. While the idea of using system characteristics as a comparative lens originates with Bresniker et al., we adapt and focus it on six characteristics aligned with the security, elasticity, and heterogeneity requirements from our threat model. Applying this lens in a structured qualitative comparison reveals design gaps unaddressed by existing systems. The following subsections present these characteristics (§3.1) and our comparative analysis (§3.2).

3.1 Criteria

The six characteristics are: **Distributed Structure:** Whether capability checks are enforced at both memory and compute nodes; *fully* distributed if enforcement occurs at both ends and intra-node delegation is handled locally, *partially* if only on the memory side. **Subject Granularity:** The smallest subject unit distinguished; full support means isolating individual processes at the compute node. **Object Granularity:** The smallest controllable unit of a resource; for memory, the byte level enables the finest isolation. **Capability Delegation:** A process may delegate capability privileges only through authorized channels. **Capability Revocation:** A process can revoke a delegated capability and all its re-delegations. **Persistency:** Capabilities survive process termination; in persistent resources such as non-volatile memory, this prevents residual access after reboot.

3.2 Results

Our review covered nineteen capability systems (Table 1) relevant to capability enforcement in resource-disaggregated or comparable distributed settings. While many support some characteristics individually, simultaneous support is rare due to trade-offs, and none fully supports all six. About half provide distributed enforcement. Subject and object granularity are most common, with all systems offering at least partial support. Delegation, revocation, and persistency are supported—fully or partially—by seventeen (89%), eleven (57%), and eleven (57%) systems, respectively, but seldom together, limiting secure, dynamic cooperation in elastic pools. In resource-disaggregated architectures, distributed enforcement is essential; its absence greatly limits applicability. Only five distributed systems cover all six characteristics even partially, and of these, only two—CEP and SemperOS—offer fine-grained object granularity. We therefore focus on these two as state-of-the-art baselines for our comparison (§3.3).

3.3 State-of-the-art and Comparison

CEP. CEP is a connection-oriented capability system in which a single-process OS on a CHERI-enabled microcontroller enforces access control on the memory side. Its authors [5] report that handling each process with a dedicated thread can create scalability and denial-of-service risks if the compute-side OS is untrusted. They also note that relying on the client OS for process–channel binding allows a compromised OS to impersonate local processes, and that address-space compartmentalization is absent at the compute node. CEP uses fixed-size per-process capability tables with ephemeral handles, leaving table exhaustion and extensibility to future work; preserving handles across `fork` requires cloning the table without cost evaluation. Revocation is centralized at the memory-side controller by traversing the derivation graph, which the authors identify as a scalability concern. SADRA avoids these limitations by using a connection-less hardware design (FPGA or ASIC) with fully distributed enforcement at both compute and resource nodes, compute-side compartmentalization, an elastic capability tree, efficient `fork` handling, message-free revocation, local capability minting, and SmartNIC-embedded per-principal policy enforcement, evaluated on datacenter-grade FPGA SmartNICs.

SemperOS. SemperOS is a capability-based OS for disaggregated environments. Its authors [26] state that capabilities are statically assigned to compute nodes, preventing migration of new nodes at runtime. They also describe a revocation mechanism for inter-node delegated capabilities that requires inter-kernel messaging, making the system vulnerable to denial-of-service attacks via long delegation chains. Elasticity and dynamic reconfiguration in heterogeneous pools are addressed only partially; for example, stale capabilities are not systematically handled. SADRA overcomes these limitations by supporting dynamic node migration and capability assignment, performing revocation entirely within the enforcement layer without messaging, unifying capability storage and delegation tracking in a scalable tree, and enabling full elasticity in heterogeneous resource pools.

SADRA's primary design goals are to be fully distributed, provide elasticity, and support heterogeneous resource pools—characteristics identified in our review as critical for overcoming the trade-offs in existing designs. As shown in §4, it fills these gaps by supporting all six characteristics while avoiding the performance and scalability limitations of prior systems. SADRA integrates these functional goals with the key security objectives from our threat model (§2.3), ensuring that subjects cannot forge or obtain capabilities except through controlled mechanisms, nor delegate rights they do not possess. By uniting these functional and security properties in a SmartNIC-based enforcement architecture, SADRA achieves practical deployability with strong security guarantees for resource-disaggregated architectures.

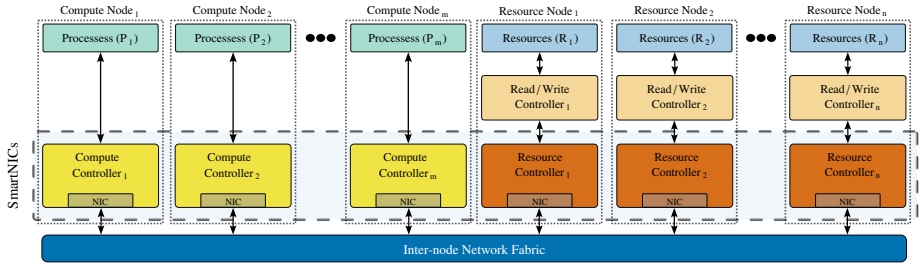


Figure 4: SADRA capability-based access control design.

4 Proposed Access-Control System

This section presents the design of SADRA, a capability-based access control system for resource-disaggregated architectures. Unlike prior systems that rely on host software or split enforcement, SADRA confines all logic to SmartNICs, removing the host from the trusted base. We formalize disaggregated systems to define components, interactions, and enforcement points, introduce distributed controllers across compute and resource nodes, and present a capability model that specifies how rights are represented, delegated, and revoked with fine-grained enforcement, achieving low latency as shown in **??**.

4.1 Abstract Model of a Disaggregated System

We construct an abstract model of resource-disaggregated architectures to capture their structure and identify the entities and interactions relevant to capability-based enforcement. As illustrated in Figure 1, the model consists of resource nodes, compute nodes, and an inter-node network fabric. Resource nodes manage hardware resources and operate independently, each with a read/write controller responsible for processing requests. Compute nodes host processes that interact with these controllers using a request–response pattern.

Formally, we define the model as $\mathcal{M} = (\mathbb{N}_r, \mathbb{N}_c, \mathbb{R}, \mathbb{P}, F)$, where $\mathbb{N}_r$ is the set of resource nodes, $\mathbb{N}_c$ the set of compute nodes, $\mathbb{R}$ the set of resources, $\mathbb{P}$ the set of processes, and F the inter-node network fabric. Each resource node $N_r^j \in \mathbb{N}_r$ $(1 \leqslant j \leqslant |\mathbb{N}_r|)$ manages a subset of resources $R_j \subseteq \mathbb{R}$ and includes a read/write controller, while each compute node $N_c^k \in \mathbb{N}_c$ $(1 \leqslant k \leqslant |\mathbb{N}_c|)$ hosts a subset of processes $P_k \subseteq \mathbb{P}$. This abstract model forms the basis for reasoning about access control, delegation, and enforcement in our system design.

4.2 Distributed Access Controllers

In our model, access controllers mediate requests between processes and resources. A *resource controller* $RC_j \in \mathbb{RC}$ resides in resource node N_r^j and enforces policies on its resources. A *compute controller* $CC_k \in \mathbb{CC}$ resides in compute node N_c^k and mediates process requests to remote resources. This two-level placement enforces policies

at both source and destination, avoids bottlenecks, and supports scalable enforcement. Controllers of the same type operate independently, without inter-controller communication.

Compute controllers perform two main functions. First, they execute the first stage of access control at the source, verifying that the requesting process holds the necessary rights before the request leaves the compute node; invalid requests are denied immediately. Second, they manage intra-node delegation and revocation, allowing rights to be transferred between co-located processes without involving a remote resource controller, thus avoiding network round trips. For example, if two processes $p, p' \in P_k$ reside on the same compute node and p has access to a resource $r \in R_j$, the compute controller CC_k can delegate this right to p' directly.

Resource controllers perform three main functions: resource allocation, inter-node delegation and revocation of access rights, and access control enforcement. First, they assign resources in their node to processes through allocation. Second, they enable delegation and revocation when the delegating and receiving processes run on different compute nodes. For example, if $p \in P_k$ requests to share its privilege over resource $r \in R_j$ with $p' \in P_{k'}$, the resource controller RC_j coordinates with compute controllers CC_k and $CC_{k'}$ to process delegation. Finally, they execute the second stage of access control at the destination. Continuing the example, RC_j will deny p' access to r if p has revoked the corresponding permission. All tasks are performed using capabilities, ensuring fine-grained and verifiable rights management.

Next, we present the capability model that specifies how these rights are represented, delegated, and revoked across controllers.

4.3 Capability Model

To enforce distributed access control without relying on host software, we design a capability model that unifies representation, delegation, and revocation across compute and resource controllers. Our model enables resource controllers to delegate certain operations to compute controllers while ensuring that all enforcement is mediated through capabilities. Resource controllers accept requests only from compute controllers, which intercept process requests at the source. This arrangement guarantees that every request is checked twice—once at the compute controller and once at the resource controller—before reaching the target resource.

The model defines three capability types: *resource* ($\mathbb{C}_r$), *compute* ($\mathbb{C}_c$), and *process* ($\mathbb{C}_p$), as illustrated in Figure 5. Each capability is tied to its controller or process and encodes a process's rights to a resource. Process and compute capabilities also carry a pointer to the next capability in the enforcement chain, denoted $\mapsto$. For example, if $c_p \mapsto c_c$ and $c_c \mapsto c_r$, with $c_p \in \mathbb{C}_p$, $c_c \in \mathbb{C}_c$, and $c_r \in \mathbb{C}_r$, then all three grant identical permissions to resource $r \in R_j$.

4.3.1 Indicator Flag

To revoke a delegated capability, a process must identify the specific capability instance that it previously granted to another principal. We refer to the capability used to perform the delegation as the *delegator capability*, and to the capability generated for

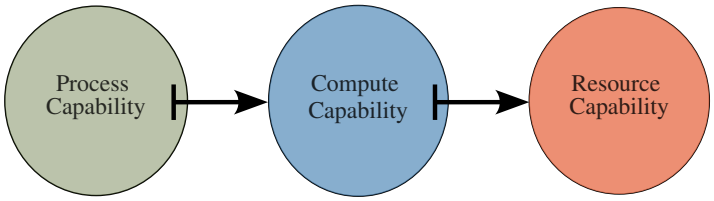


Figure 5: Capability-object model with process ($\mathbb{C}_p$), compute ($\mathbb{C}_c$), and resource ($\mathbb{C}_r$) capabilities. Arrows show the $\mapsto$ relation: process $\rightarrow$ compute $\rightarrow$ resource, each set holding identical rights to the same $r \in R_j$ for enforcement at compute and resource controllers.

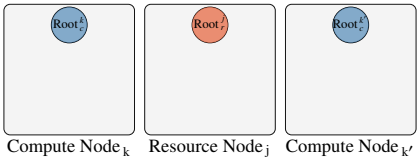
the recipient as the *delegated capability*. SADRA tracks delegated capabilities using a boolean *indicator flag* stored within each capability object. When set, this flag marks the capability as an **indicator**, which points to the corresponding delegated capability. Indicators are used solely for delegation tracking and cannot be used to perform read, write, or delegation operations; any such attempt is rejected by the enforcement logic. We explain how SADRA leverages indicators during capability-related operations in §4.3.3. The complete formal definitions of indicators is provided in §4.4. To efficiently store, manage, and traverse capabilities, SADRA organizes them into a hierarchical data structure called the capability tree.

4.3.2 Capability Tree

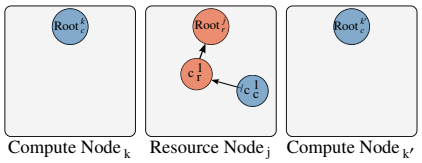
Controllers store their capabilities in *rooted trees* [49], hierarchical structures with a designated root capability. We call such a structure a *capability tree* ($T \in \{\mathbb{T}_r, \mathbb{T}_c\}$) when its nodes are capabilities. In a resource capability tree, each edge corresponds to a delegation from parent to child. In a compute capability tree, the root is a *virtual capability* that serves as an anchor for multiple subtrees received from different resource controllers; all other edges follow the delegation relation. Controllers add or remove capabilities in their trees to perform allocation, delegation, and revocation. Capability trees are stored in non-volatile memory, ensuring persistence across reboots. Resource and compute capabilities outlive their corresponding process capabilities, allowing controllers to regenerate them when needed. A complete formal definition of capability trees is given in §4.4.

4.3.3 Capability Operations

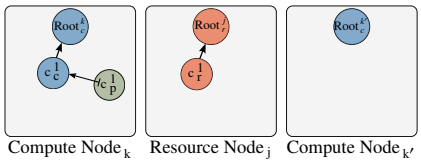
Capability operations in SADRA include resource allocation, delegation, and revocation. We illustrate these with an example: process $p \in P_k$ requests *read/write* access to resource $r \in R_j$ (*allocation*). After obtaining access, p delegates the rights to process $p' \in P_k$ on the same compute node (*intra-node delegation*) and to process $p'' \in P_{k'}$ on a different compute node (*inter-node delegation*). Finally, p revokes both delegations (*intra-node* and *inter-node revocation*). Figures 6a to 6i show how resource and compute controllers update their capability trees to execute these operations. Initially, each controller's tree contains only the root capability (Figure 6a).



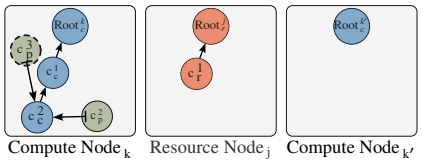
(a) Initial state.



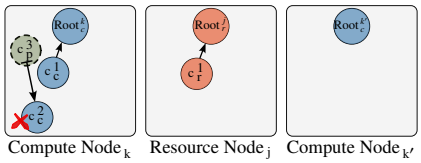
(b) Resource allocation - Resource Node.



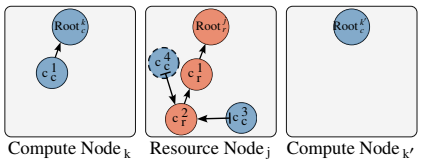
(c) Resource allocation - Compute Nodes.



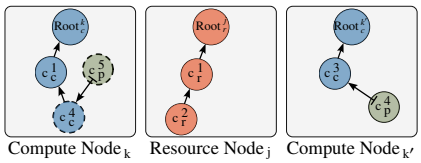
(d) Intra-Node Delegation.



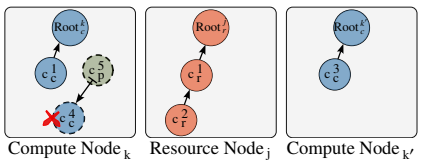
(e) Intra-Node Revocation.



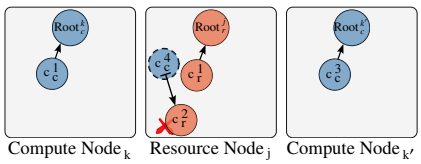
(f) Inter-Node Delegation - Resource Node.



(g) Inter-Node Delegation - Compute Nodes.



(h) Inter-Node Revocation - Compute Nodes.



(i) Inter-Node Revocation - Resource Node.

Figure 6: Capability Trees. Legend: ● = resource capability; ● = intermediate capability; ● = process capability;
○ = indicator flag is unset; ◌ = indicator flag is set; ⟼: point-to connection in our capability-object model;
⟶: child-parent relation in a capability tree.

Resource Allocation. When process p requests access to resource r, RC_j allocates r by creating a resource capability c_r^1 encoding p's rights. It adds c_r^1 as a child of the root in $T_j \in \mathbb{T}_r$, derives compute capability c_c^1 with $c_c^1 \mapsto c_r^1$, and sends c_c^1 to CC_k

(Figure 6b). CC_k adds c_c^1 to its tree $T_k \in \mathbb{T}_c$, derives process capability c_p^1 with $c_p^1 \mapsto c_c^1$, and returns c_p^1 to p (Figure 6c). This capability serves as p's unforgeable authority to access r.

Intra-Node Delegation. To share rights locally, CC_k creates compute capability c_c^2, inserts it as a child of c_c^1 in T_k, derives process capability c_p^2, and sends it to p', enabling access to r (Figure 6d). It also issues indicator capability c_p^3 for p, as defined in §4.3.1, enabling later revocation. Indicators only track delegation and cannot authorize requests.

Intra-Node Revocation. To revoke p''s access, p invokes c_p^3. CC_k resolves it to c_c^2 in T_k, deletes c_c^2, and thus invalidates c_p^2 (Figure 6e). Since $c_p^2 \mapsto c_c^2$ now points to a removed entry, subsequent attempts by p' are denied.

Inter-Node Delegation. For cross-node sharing, CC_k collaborates with RC_j. RC_j creates resource capability c_r^2, inserts it as a child of the root in T_j, and derives c_c^3 for $CC_{k'}$ and c_c^4 for CC_k (Figure 6f). Both compute controllers insert the received capabilities (Figure 6g). $CC_{k'}$ derives process capability c_p^4 for p'', while CC_k issues indicator c_p^5 for p.

Inter-Node Revocation. To revoke p'''s access, p uses c_p^5. CC_k resolves it to c_c^4, removes it, and forwards c_c^4 to RC_j (Figure 6h). RC_j resolves c_c^4 to c_r^2, deletes c_r^2, and invalidates all dependent capabilities (Figure 6i).

A subsequent request from p'' using c_p^4 still passes at $CC_{k'}$, since $c_p^4 \mapsto c_c^3$ remains in $T_{k'}$. However, it fails at RC_j because $c_c^3 \mapsto c_r^2$ no longer exists. $CC_{k'}$ then removes c_c^3 and reports the failure to p''. Formal semantics of allocation, delegation, and revocation are as follows.

4.4 Formal Definitions

We formalize the abstractions used throughout SADRA's design and security proof. Our model consists of processes, resources, and capabilities, which together describe how subjects obtain and exercise rights over objects. These definitions form the basis for reasoning about allocation, delegation, and revocation.

Definition 1 (Process)**.** *Let $\mathbb{P}$ denote the universe of processes. Each process $p \in \mathbb{P}$ executes on a compute node and may request, hold, or delegate capabilities.*

We next define the objects over which processes hold rights.

Definition 2 (Resource)**.** *Let $\mathbb{R}$ denote the universe of resources (e.g., memory, storage, accelerators). A resource $r \in \mathbb{R}$ is an object over which processes may exercise rights.*

Capabilities capture the relation between subjects and resources.

Definition 3 (Capability)**.** *A* capability *is a tuple*

$$c = \langle s, r, \mathcal{R}, \tau \rangle$$

where $s \in \mathbb{P}$ is the subject (process), $r \in \mathbb{R}$ is the resource, $\mathcal{R} \subseteq \{\mathsf{read}, \mathsf{write}, \mathsf{exec}, \dots\}$ is the set of rights, and τ is metadata (e.g., delegation depth, expiration, or information-flow label).

To formalize capability operations, we introduce a simple indicator predicate.

Definition 4 (Capability Indicator)**.** *We write*

$$\mathsf{HasCap}(p, r, \alpha) = 1$$

if process $p \in \mathbb{P}$ holds a capability for resource $r \in \mathbb{R}$ that includes right $\alpha \in \{\mathsf{read}, \mathsf{write}, \mathsf{exec}, \dots\}$. If no such capability exists, $\mathsf{HasCap}(p, r, \alpha) = 0$.

We now capture delegation structure with a tree over derived capabilities.

Definition 5 (Delegation Derivation)**.** *Given capabilities $c = \langle s, r, \mathcal{R}, \tau \rangle$ and $c' = \langle s', r, \mathcal{R}', \tau' \rangle$, we write $c' \preceq c$ if c' is derived from c by delegation with $\mathcal{R}' \subseteq \mathcal{R}$ and τ' updated according to policy.*

This yields a rooted tree for each seed capability.

Definition 6 (Capability Tree)**.** *For a seed capability c_0, the capability tree is $T(c_0) = \langle V, E, \text{root} \rangle$ where $V = \{c \mid c \preceq^* c_0\}$,* root $= c_0$*, and $E = \{(c, c') \mid c' \preceq c\}$.*

Edges must preserve safety via monotonic rights.

Definition 7 (Monotonicity)**.** *If $(c, c') \in E$ with $c = \langle s, r, \mathcal{R}, \tau \rangle$ and $c' = \langle s', r, \mathcal{R}', \tau' \rangle$, then $\mathcal{R}' \subseteq \mathcal{R}$ and τ' satisfies the delegation policy (e.g., nonincreasing depth or expiry).*

Revocation removes subtrees and defines which capabilities remain usable.

Definition 8 (Revocation and Validity)**.** *Let* $\mathsf{Revoked} \subseteq V$ *mark nodes selected for revocation; define* $\mathsf{Subtree}(c^*) = \{c \in V \mid c^* \preceq^* c\}$ *and* $\mathsf{Valid}(T) = V \setminus \bigcup_{c^* \in \mathsf{Revoked}} \mathsf{Subtree}(c^*)$.

Capability-Related Operations

We now formalize the operations that govern capabilities in SADRA. These include allocation, delegation, and revocation, which together describe how capabilities are created, propagated, and invalidated. Locality matters only for delegation, which may occur within a compute node or across nodes, and enforcement depends jointly on compute- and resource-side validity.

Definition 9 (Node Mapping)**.** *Let $\mathbb{K}$ be the set of compute nodes and* $\text{node} : \mathbb{P} \to \mathbb{K}$ *map each process to its compute node.*

Allocation creates seed capabilities and is locality-independent.

Definition 10 (Allocation)**.** *An allocation grants process $p \in \mathbb{P}$ rights $\mathcal{R}$ over resource $r \in \mathbb{R}$ by creating a seed capability $c_0 = \langle p, r, \mathcal{R}, \tau \rangle$ and setting* $\mathsf{HasCap}(p, r, \alpha) = 1$ *for all $\alpha \in \mathcal{R}$.*

Delegation within the same compute node extends rights locally.

Definition 11 (Delegation$_{\text{intra}}$)**.** *Given* $c = \langle s, r, \mathcal{R}, \tau \rangle$ *and a target process* s' *with* $\text{node}(s) = \text{node}(s')$*,*

$$c' \preceq c \iff \begin{array}{l} c' = \langle s', r, \mathcal{R}', \tau' \rangle, \\ \mathcal{R}' \subseteq \mathcal{R}, \\ \tau' \textit{ satisfies intra-node policy}. \end{array}$$

Delegation then sets $\mathsf{HasCap}(s', r, \alpha) = 1$ *for all* $\alpha \in \mathcal{R}'$*.*

Delegation across nodes propagates rights globally.

Definition 12 (Delegation$_{\text{inter}}$)**.** *Given* $c = \langle s, r, \mathcal{R}, \tau \rangle$ *and a target process* s' *with* $\text{node}(s) \neq \text{node}(s')$*,*

$$c' \preceq c \iff \begin{array}{l} c' = \langle s', r, \mathcal{R}', \tau' \rangle, \\ \mathcal{R}' \subseteq \mathcal{R}, \\ \tau' \textit{ satisfies inter-node policy}. \end{array}$$

Delegation then sets $\mathsf{HasCap}(s', r, \alpha) = 1$ *for all* $\alpha \in \mathcal{R}'$*.*

Revocation invalidates subtrees regardless of locality.

Definition 13 (Revocation)**.** *Revoking a node* c^* *removes* $\mathsf{Subtree}(c^*) = \{c \mid c^* \preceq^* c\}$ *from the tree, yielding* $\mathsf{Valid}(T)$ *and resetting* $\mathsf{HasCap}(p, r, \alpha) = 0$ *for all processes* p *and rights* α *in revoked nodes.*

We distinguish two trees that track validity from different enforcement perspectives.

Definition 14 (Compute Capability Tree)**.** *For process* p *on node* $k = \text{node}(p)$*, the compute capability tree* $T_{\mathsf{comp}}(p)$ *records all capabilities allocated to or delegated within node k, with validity* $\mathsf{Valid}(T_{\mathsf{comp}}(p))$ *defined by subtree removal.*

Definition 15 (Resource Capability Tree)**.** *For a resource* $r \in \mathbb{R}$*, the resource capability tree* $T_{\mathsf{res}}(r)$ *records all inter-node delegations rooted at seed capabilities of r, with validity* $\mathsf{Valid}(T_{\mathsf{res}}(r))$ *defined by subtree removal.*

Effective rights require agreement between compute and resource views.

Definition 16 (Combined Enforcement)**.** *For process* p *and resource* r*,*

$$\begin{aligned} &\mathsf{HasCap}(p, r, \alpha) = 1 \iff \exists\, c = \langle s{=}p, r, \mathcal{R}, \tau \rangle : \\ &\alpha \in \mathcal{R} \ \wedge \ c \in \mathsf{Valid}\Big(T_{\mathsf{comp}}(p)\Big) \ \wedge \ c \in \mathsf{Valid}(T_{\mathsf{res}}(r)) \,. \end{aligned}$$

These definitions provide a complete formal foundation for reasoning about capability operations in SADRA. In particular, they establish how rights are created, propagated, and revoked across both compute and resource views, which we build on in the following theorems to prove soundness and security properties.

5 System Design's Soundness

Now that we have seen how our system controls access, we prove that our capability-based access control system is sound. Following Maffeis et al. [41] work, we prove our system design's soundness by demonstrating that it satisfies the capability-safety and authority-safety properties. Furthermore, we prove that our system guarantees isolation property based on these two properties.

In the rest of the section, we employ a few auxiliary functions in the definitions and proofs. Table 2 lists the functions and their descriptions, in which $\mathbb{V} = \{r, w\}$ is the permission set (r: read, w: write), and $\mathbb{R}$ denotes the set of all resources.

5.1 Well-Formed Capability Trees

In SADRA, resource and compute controllers each maintain a *capability tree* to enforce delegation and revocation in a distributed setting. The correctness of these operations requires *well-formed* trees, captured by a *partial order* ($\preceq$) over capabilities. This order enforces delegation monotonicity: a capability may only delegate to one referencing a subset of its resources and a subset of its permissions. Let priv : $\mathbb{C} \to 2^{\mathbb{V}}$ return a capability's permission set, and rsrc : $\mathbb{C} \to \mathbb{R}$ return the *set* of resources it governs (e.g., memory ranges).

Definition 17 (Partial Order of Capabilities)**.** *Two capabilities c' and c satisfy $c' \preceq c$ if and only if* $\text{rsrc}(c') \subseteq \text{rsrc}(c)$ *and* $\text{priv}(c') \subseteq \text{priv}(c)$*.*

Intuitively, c' is more restricted than c. This ensures delegation chains remain monotonic across both controller-side trees, preventing privilege amplification.

To state well-formedness and the upcoming lemmas, we use the following auxiliary functions: getRoot : $\mathbb{T}_x \to \mathbb{C}_x$ (tree root), isRoot : $\mathbb{T}_x \times \mathbb{C}_x \to \mathbb{B}$ (root check), and isInTree : $\mathbb{T}_x \times \mathbb{C}_x \times \mathbb{C}_x \to \mathbb{B}$ (membership starting from a node).

5.1.0.1 Resource-Capability Tree

A resource-capability tree $T_r \in \mathbb{T}_r$ is a rooted tree whose root $c_r^{root} \in \mathbb{C}_r$ belongs to the resource controller. The root capability references all resources managed by that controller (e.g., a memory range or device partition) and grants full permissions. In a disaggregated architecture, each subtree represents a delegation hierarchy enforced by the SmartNIC, and well-formedness ensures these hierarchies preserve monotonic authority.

Intuitively, every child capability must be more restricted than its parent: it can only reference a subset of the resources (e.g., a smaller memory range) and grant a subset of the permissions (e.g., read-only instead of read/write). This ensures delegation always flows from stronger to weaker authority.

For example, if c_c^1 grants *read/write* access to memory range $[0, 100)$, a delegated capability c_c^2 might grant only *read* access to $[25, 50)$. This narrowing of both the resource set and the permission set preserves the partial order (see Figure 6d).

Definition 18 (Well-Formed Resource-Capability Tree)**.** *A tree T_r is well-formed iff for all c_r, $c'_r \in \mathbb{C}_r$, if* $\text{isInTree}(T_r, c_r, c'_r)$ *holds, then $c'_r \preceq c_r$.*

Theorem 5.1. *If a resource-capability tree $T_r \in \mathbb{T}_r$ is well-formed, then it remains well-formed under all capability operations (allocation, delegation, and revocation).*

This invariant is non-trivial: without it, delegation could create cycles, amplify authority, or leave stale capabilities. SADRA's enforcement guarantees that all capability operations preserve the partial order.

Proof. We make the well-formedness predicate $\mathsf{WF}_{\mathsf{res}}(T_{\mathsf{res}}(r))$ explicit via:

(WF0) **Root invariant:** There is a distinguished root $c_0 = \langle s_0, r, \mathcal{R}_0, \tau_0 \rangle$ with subject s_0 at the resource controller; c_0 has no parent and is not revocable.

(WF1) **Tree shape:** $T_{\mathsf{res}}(r)$ is a rooted directed tree (acyclic; each $c \neq c_0$ has a unique parent).

(WF2) **Resource consistency:** Every node $c = \langle s, r', \mathcal{R}, \tau \rangle$ has $r' = r$.

(WF3) **Monotonic rights:** For every edge $(c, c') \in E$, if $c = \langle s, r, \mathcal{R}, \tau \rangle$ and $c' = \langle s', r, \mathcal{R}', \tau' \rangle$ then $\mathcal{R}' \subseteq \mathcal{R}$.

(WF4) **Policy compliance:** For every edge $(c, c') \in E$, the metadata update (τ, τ') satisfies the inter-node policy (e.g., nonincreasing depth/expiry).

(WF5) **Inter-node edges:** For every edge $(c, c') \in E$, $\mathrm{node}(s) \neq \mathrm{node}(s')$.

(WF6) **Freshness:** Any inserted node is fresh ($c' \notin V$ prior to insertion).

Note. Intra-node delegation does not modify $T_{\mathsf{res}}(r)$ and is thus a no-op here.

We prove preservation by a one-step case analysis. Assume $\mathsf{WF}_{\mathsf{res}}(T_{\mathsf{res}}(r))$.

Case 1 — Allocation. The controller creates $c = \langle s, r, \mathcal{R}, \tau \rangle$ for some process s and inserts the edge (c_0, c). By (WF6) c is fresh; adding a leaf preserves acyclicity and unique parent (WF1). Nodes are labeled with r (WF2). Root rights give $\mathcal{R} \subseteq \mathcal{R}_0$ (WF3), and the update (τ_0, τ) respects policy (WF4). Since s_0 is at the resource controller and s is on a compute node, $\mathrm{node}(s_0) \neq \mathrm{node}(s)$ (WF5). (WF0) still holds because c_0 is unchanged.

Case 2 — Inter-node delegation. Given parent $c = \langle s, r, \mathcal{R}, \tau \rangle$ and a target process s' on a different node, the controller creates $c' = \langle s', r, \mathcal{R}', \tau' \rangle$ with $\mathcal{R}' \subseteq \mathcal{R}$ and inserts (c, c'). Fresh insertion preserves the tree shape (WF1); labels keep r (WF2). Monotonicity and policy compliance hold by construction (WF3–WF4). Since $\mathrm{node}(s) \neq \mathrm{node}(s')$, (WF5) holds. Root invariant (WF0) is unaffected.

Case 3 — Revocation. Removing a node $c^* \neq c_0$ and all nodes in its subtree deletes a connected set of edges. The remainder remains a rooted tree (WF1); surviving nodes/edges are unchanged, so (WF2)–(WF5) remain true. (WF0) holds because the root is not removed. Freshness (WF6) is not relevant to deletions.

In each case, $\mathsf{WF}_{\mathsf{res}}(T'_{\mathsf{res}}(r))$ holds. By iterating the one-step argument, preservation holds for any finite sequence of such operations. □

5.1.0.2 Compute-Capability Tree

A compute-capability tree $T_c \in \mathbb{T}_c$ is a rooted tree whose root $c_c^{root} \in \mathbb{C}_c$ belongs to the compute controller. The root capability serves as a structural anchor: it references no resources and grants no permissions, but ensures that every delegation hierarchy has a common parent. Tree T_c is *well-formed* if every subtree (delegation hierarchy), excluding the root, respects the partial-order relation:

Definition 19 (Well-Formed Compute-Capability Tree)**.** *A tree T_c is well-formed if and only if for all c_c, $c'_c \in \mathbb{C}_c$, if* $\text{isInTree}(T_c, c_c, c'_c)$ *and* $\neg\, \text{isRoot}(T_c, c_c)$ *hold, then $c'_c \preceq c_c$.*

Theorem 5.2. *If a compute-capability tree $T_c \in \mathbb{T}_c$ is well-formed, then it remains well-formed after any capability-related operation.*

Intuitively, delegation from non-root nodes only creates descendants with restricted rights, so the partial-order constraint is preserved. Formal proofs of Theorem 5.1 and Theorem 5.2 appear in **??**.

Proof. We spell out $\mathsf{WF}_{\mathsf{comp}}(T_{\mathsf{comp}}(k))$ via:

(CF0) **Dummy root:** distinguished root $\hat{c}_k$ (no rights/subject), no parent, not revocable.

(CF1) **Tree shape:** rooted directed tree (acyclic; each $c \neq \hat{c}_k$ has a unique parent).

(CF2) **Local nodes:** every $c = \langle p, r, \mathcal{R}, \tau \rangle$ has $\text{node}(p) = k$.

(CF3) **Intra-node edges:** for each edge (c, c'), subjects are local to k.

(CF4) **Monotonic rights:** for edge (c, c'), if $c = \langle p, r, \mathcal{R}, \tau \rangle$ and $c' = \langle p', r, \mathcal{R}', \tau' \rangle$, then $\mathcal{R}' \subseteq \mathcal{R}$.

(CF5) **Intra-node policy:** metadata update (τ, τ') satisfies intra-node policy.

(CF6) **Freshness:** any inserted node c' was not in V before insertion.

Assume $\mathsf{WF}_{\mathsf{comp}}(T_{\mathsf{comp}}(k))$ and consider one modifying operation:

Allocation. CC_k creates $c = \langle p, r, \mathcal{R}, \tau \rangle$ for some local p and inserts $(\hat{c}_k, c)$. Fresh leaf $\Rightarrow$ (CF1, CF6); locality and intra-node edge $\Rightarrow$ (CF2, CF3); monotone rights/policy by construction $\Rightarrow$ (CF4, CF5); (CF0) unchanged.

Receive-from-external delegation. Remote node delegates to local p; CC_k inserts fresh c as child of $\hat{c}_k$. Same argument as allocation: (CF1–CF3, CF4–CF6) hold; (CF0) unchanged.

Intra-node delegation. Given $c = \langle p, r, \mathcal{R}, \tau \rangle$ and target p' local to k, create $c' = \langle p', r, \mathcal{R}', \tau' \rangle$ with $\mathcal{R}' \subseteq \mathcal{R}$ and insert (c, c'). Fresh leaf preserves (CF1, CF6); locality/edge (CF2, CF3); monotonicity/policy (CF4, CF5); (CF0) unchanged.

Intra-node revocation. Remove $c^* \neq \hat{c}_k$ and its subtree. Remainder is a rooted tree with same root (CF1, CF0); surviving nodes/edges unchanged (CF2–CF5); (CF6) irrelevant.

Hence $\mathsf{WF}_{\mathsf{comp}}(T'_{\mathsf{comp}}(k))$ holds after any one operation; by iteration, after any finite sequence. □

Table 2: Function Definitions

Function Name	Function Signature	Description
rsrc	$\mathbb{C} \to \mathbb{R}$	returns the pointed resources inside a capability.
priv	$\mathbb{C} \to 2^{\mathbb{V}}$	returns the permissions inside a capability
getRoot	$\mathbb{T}_x \to \mathbb{C}_x$	returns the root of a capability tree.
isRoot	$\mathbb{T}_x \times \mathbb{C}_x \to boolean$	checks if a capability is the root of a capability tree
isInTree	$\mathbb{T}_x \times \mathbb{C}_x \times \mathbb{C}_x \to boolean$	checks if the second capability is inside a tree, where the first capability indicates the search's start point.

Legend: $x \in \{r, i\}$, $\mathbb{C} = \{\mathbb{C}_r \cup \mathbb{C}_i \cup \mathbb{C}_p\}$

5.2 Valid Capability

We define capability validity in terms of well-formed capability trees. A capability is valid if it can be transitively linked back to a trusted root forged by a resource controller. This yields two cases: (1) compute capabilities, which must map to valid resource capabilities; and (2) process capabilities, which must resolve to an *original* compute capability issued by a resource controller. Controllers enforce these invariants before granting access or honoring delegations.

5.2.0.1 Valid Compute Capability.

A compute capability c_c is valid if a resource controller has forged it and it points to an existing resource capability in that controller's tree. For example, c_c^1 in Figure 6i points to c_r^1, making it valid, whereas c_c^3 points to a removed capability and is invalid.

Definition 20 (Valid Compute Capability)**.** *A compute capability $c_c \in \mathbb{C}c$ is valid, i.e.,* isValidCompCap(c_c) *holds, if there exist $c_r \in \mathbb{C}r$ and $Tr \in \mathbb{T}r$ such that $c_c \mapsto c_r$ and* isInTree(T_r, getRoot(T_r), c_r).

5.2.0.2 Valid Process Capability.

A process capability c_p is valid if it points to a compute capability in a well-formed tree that resolves to an *original* node issued by a resource controller. For example, in T_c^k of Figure 6d, c_c^1 is original while c_c^2 is delegated. If a process holds c_p^2, controller IC_k must forward its anchor c_c^1 instead, preserving the invariant.

We define *getOriginalCap* : $\mathbb{T}_c \times \mathbb{C}_c \to \mathbb{C}_c$ as the function that, given a compute capability, traverses toward the root until it reaches its original capability.

Definition 21 (Valid Process Capability)**.** *A process capability c_p is valid, i.e.,* isValidProCap(c_p) *holds, if there exist $c_c, , c_c' \in \mathbb{C}c$ and $Tc \in \mathbb{T}c$ such that $c_p \mapsto c_c$,* isInTree(Tc, getRoot(T_c), c_c), c_c' = getOriginalCap(T_c, c_c)*, and* isValidCompCap(c_c').

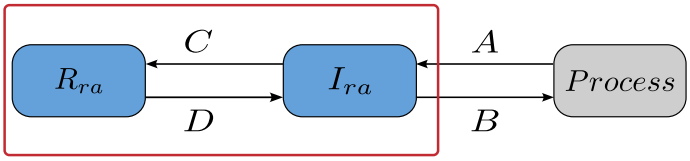


Figure 7: Syntactic Interfaces of the Resource-allocation Programs.

5.2.0.3 Summary.

By anchoring compute and process capabilities in resource-controller–forged roots, SADRA ensures all capabilities reflect legitimate allocations or delegations—an invariant that underpins the capability-safety property. We next introduce the notion of a *capability system* to formalize the state and operations required for stating security properties.

5.3 Security Properties

Our access-control system guarantees three security properties, including: *capability safety*, *authority safety*, and *isolation*. We prove our system satisfies these properties by demonstrating that our design handles capability-related operations correctly; thus, we first define five lemmas, one for each operation, and verify them using the assumption-commitment technique.

Our modeling has three component types: *process*, *resource controller*, and *intermediate controller*. In this modeling, the components interact to execute an operation. We consider five operations in this modeling: resource allocation, internal capability delegation, external capability delegation, internal capability revocation, and external capability revocation. In addition, we define five lemmas based on the operations and explain the operations' precondition, postcondition, assumption, and commitment properties to prove the lemmas based on the assumption-commitment (A-C) technique. Each proof considers the states of the components involved in the corresponding operation.

5.4 First Lemma: Resource Allocation

The first lemma indicates that any resource-allocation operation creates a valid p-cap. We formally define the lemma and proof it.

Lemma 1. *Given a process with a unique PID in a process node with a unique NID, an intermediate controller in the same node, and an resource controller in a resource node with a unique NID, the process will get a valid p-cap by sending a resource-allocation request.*

Proof. We use the assumption-commitment technique to prove the lemma.

To prove the lemma, we model the intermediate and the resource controllers in a resource-allocation operation by programs $I_{ra} \stackrel{\text{def}}{=} (L^i_{ra}, T^i_{ra}, s^i_{ra}, t^i_{ra})$ and $R_{ra} \stackrel{\text{def}}{=} (L^r_{ra}, T^r_{ra}, s^r_{ra}, t^r_{ra})$,

respectively. The programs and the process exchange messages via synchronous channels to handle resource-allocation requests. Figure 7 depicts these two programs and their syntactic interfaces. In addition, figures 8a and 8b illustrate sequential diagrams of programs I_{ra} and R_{ra}, respectively. In the next step, we prove that when the process sends a resource-allocation request, it will receive a valid p-cap.

5.4.1 Resource Allocation - Intermediate Controller

Program I_{ra} receives a resource-allocation request from the process via channel A, creates a corresponding request for program R_{ra}, and sends it through channel C. Furthermore, It receives the response from program R_{ra} via channel D. Program I_{ra} updates its capability tree when it receives an i-cap from the resource controller by inserting the capability inside the tree. In addition, it creates the corresponding p-cap for the inserted i-cap. Finally, it creates a response for the process and sends the p-cap inside the response to the process through channel B. The functions of I_{ra} are as follows:

$$\begin{aligned}
init^i_{ra}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(T_i))),\\
f_{i,1}(\sigma) &= (\sigma : y_i \mapsto genRaReq(\sigma(x_i))),\\
f_{i,2}(\sigma) &= (\sigma : pCap \mapsto insert(\sigma(rootPtr), \sigma(z_i.cap)))\\
f_{i,3}(\sigma) &= (\sigma : w_i \mapsto genRaRes(\sigma(x_i), \sigma(pCap)))
\end{aligned}$$

The A-C formula for program I_{ra} is as follows:

$$\vdash \langle Ass^i_{ra},\ C^i_{ra}\rangle\ \{\varphi^i_{ra}\}\ I_{ra}\ \{\psi^i_{ra}\}$$

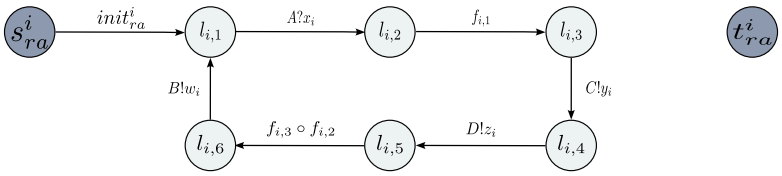
where the formal definition of φ^i_{ra}, ψ^i_{ra}, Ass^i_{ra}, and C^i_{ra} are as follows:

$$\begin{aligned}
\varphi^i_{ra} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0\ \wedge\ |icTree| \geqslant 1\\
\psi^i_{ra} &\stackrel{\text{def}}{=} false\\
Ass^i_{ra} &\stackrel{\text{def}}{=} (\#A > 0 \rightarrow\\
&\quad isRaReq(last(A))\ \wedge\ isUniquePID(last(A).spid)) \wedge\\
&\quad (\#C = \#D > 0 \rightarrow\\
&\quad isRaRes(last(D))\ \wedge\ isValidCap(last(D).cap))\\
C^i_{ra} &\stackrel{\text{def}}{=} (\#A = \#B = \#C = \#D > 0 \rightarrow\\
&\quad isRaRes(last(B))\ \wedge\ isValidCap(last(B).cap))) \wedge\\
&\quad (\#C > 0 \rightarrow\\
&\quad isRaReq(last(C))\ \wedge\ isUniquePID(last(C).spid)))
\end{aligned}$$

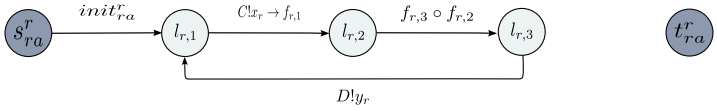
Wherein $icTree$ is the capability tree of the intermediate controller.

The assertion network for I_{ra} is as follows:

$$\begin{aligned}
Q_{s^i_{ra}} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0\ \wedge\ |T_i| \geqslant 1\\
Q_{l_{i,1}} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D
\end{aligned}$$



(a) Program I_{ra}.



(b) Program R_{ra}.

Figure 8: Resource-allocation Programs.

$$
\begin{aligned}
Q_{l_{i,2}} &\stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \\
&\quad isRaReq(last(A)) \ \wedge \ isUniquePID(last(A).spid) \\
Q_{l_{i,3}} &\stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \\
&\quad isRaReq(y_i) \ \wedge \ isUniquePID(y_i.spid) \\
Q_{l_{i,4}} &\stackrel{\text{def}}{=} \#B = \#D = (\#A - 1) = (\#C - 1) \ \wedge \ last(A) = x_i \ \wedge \\
&\quad last(C) = y_i \\
Q_{l_{i,5}} &\stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \\
&\quad last(C) = y_i \ \wedge \ last(D) = z_i \ \wedge \ isRaRes(last(D)) \ \wedge \\
&\quad isValidCap(last(D).cap) \\
Q_{l_{i,6}} &\stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \\
&\quad last(C) = y_i \ \wedge \ last(D) = z_i \ \wedge \\
&\quad isRaRes(w_i) \ \wedge \ isValidCap(w_i.cap) \\
Q_{t^i_{ra}} &\stackrel{\text{def}}{=} false
\end{aligned}
$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption A^i_{ra}, commitment C^i_{ra}, and channels A, B, C, and D.

- $\models Q_{s^i_{ra}} \to C^i_{ra}$ follows from the above definitions.
- $\models Q_{s^i_{ra}} \ \wedge \ A^i_{ra} \to Q_{l_{i,1}} \circ init^i_{ra}$. In this internal transition, the size of the intermediate-capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{i,1}} \ \wedge \ A^i_{ra} \to ((A^i_{ra} \to Q_{l_{i,2}}) \ \wedge \ C^i_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, \ h \mapsto v, \ \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Because $\#A > 0$, the

first implication of Ass_i satisfies $isRaReq(last(A))$ and $isUniquePID(last(A).spid)$. Thus, this verification holds.

- $\models Q_{l_{i,2}} \wedge A^i_{ra} \rightarrow Q_{l_{i,3}} \circ f_{i,1}$. In this internal transition, function $f_{i,1}$ creates a resource-allocation request for program R_{rr} and assigns it to variable y_i, such that $isRaReq(y_i) = true$ and $y_i.spid = last(A).spid$. In addition, $\sigma' \models isUniquePID(y_i.spid)$ because , from A^i_{ra}, we have $isUniquePID(last(A).spid) = true$. Thus, this verification holds.

- $\models Q_{l_{i,3}} \wedge A^i_{ra} \rightarrow ((A^i_{ra} \rightarrow Q_{l_{i,4}}) \wedge C^i_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y_i)))$. In this output transition, program I_{ra} sends y_i through channel C. C^i_{ra} holds because $last(C) = \sigma(y_i)$ and $\sigma \models isRaReq(y_i) \wedge isUniquePID(y_i.spid)$. Hence, this verification holds.

- $\models Q_{l_{i,4}} \wedge A^i_{ra} \rightarrow ((A^i_{ra} \rightarrow Q_{l_{i,5}}) \wedge C^i_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z_i,\ h \mapsto v,\ \sigma(h).(D, v))$ for arbitrary value v. In this input transition, a communication through channel D just took place, and function g assigns the received value to z_i. Since $\#D > 0$, from A^i_{ra} we have $\sigma' \models isRaRes(z_i) \wedge isValidCap(z_i.cap)$ Thus, this verification holds.

- $\models Q_{l_{i,5}} \wedge A^i_{ra} \rightarrow Q_{l_{i,6}} \circ (f_{i,3} \circ f_{i,2})$. In this internal transition, function $f_{i,2}$ inserts the intermediate capability $iCap$ into T_i and returns $pCap$ that points to $iCap$ inside the tree. Hence, $\sigma' \models |T_i| > 1$. Because $\sigma \models isValidCap(iCap)$ and $pCap$ points to the inserted $iCap$ inside T_i, we have $\sigma' \models isValidCap(pCap)$. Function $f_{i,3}$ creates a response for the process and assigns it to w_i, wherein $w_i.cap = pCap$. Consequently, $\sigma' \models isRaRes(w_i) \wedge isValidCap(w_i.cap)$. Thus, this verification holds.

- $\models Q_{l_{i,6}} \wedge A^i_{ra} \rightarrow ((A^i_{ra} \rightarrow Q_{l_{i,1}}) \wedge C^i_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(w_i)))$. In this output transition, program I_{ra} sends w_i through channel B. C^i_{ra} holds because $\sigma \models isRaRes(w_i) \wedge isValidCap(w_i.cap)$. Hence, this verification holds.

  Since $Q_{t^i_{ra}} \rightarrow \psi^i_{ra}$, the post-condition of program I_{ra} is satisfied.

5.4.2 Resource Allocation - Resource Controller

Program R_{ra} receives a request for allocating a resource from program I_{ra} via channel C. In the next step, program R_{ra} allocates the requested resource and inserts its corresponding resource capability inside its capability tree. Furthermore, it creates the corresponding intermediate capability of the resource capability. Finally, program R_{ra} creates a response, inserts the intermediate capability, and sends the response message through channel D toward program I_{ra}. The conditions and functions of R_{ra} are as follows:

$$
\begin{aligned}
init^r_{ra}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(T_r))),\\
f_{r,1}(\sigma) &= (\sigma : rCap \mapsto allocRes(\sigma(x_r))),\\
f_{r,2}(\sigma) &= (\sigma : iCap \mapsto insert(\sigma(T_r), \sigma(rootPtr),\\
&\qquad \sigma(rCap))),
\end{aligned}
$$

$$f_{r,3}(\sigma) \quad = \ (\sigma : y_r \mapsto genRaRes(\sigma(x_r), iCap))$$

where T_r is the resource-capability tree, $rCap$ is a resource capability , and $iCap$ is an intermediate capability.

The A-C formula for process R_{ra} is as follows:

$$\vdash \langle A^r_{ra}\ C^r_{ra}\rangle\ \{\varphi^r_{ra}\}\ R_{ra}\ \{\psi^r_{ra}\}$$

where the formal definition of φ^r_{ra}, ψ^r_{ra} , A^r_{ra}, and C^r_{ra} are as follows:

$$\begin{aligned}
\varphi^r_{ra} &\stackrel{\text{def}}{=} \#C = \#D = 0 \ \wedge\ |T_r| = 1\\
\psi^r_{ra} &\stackrel{\text{def}}{=} false\\
A^r_{ra} &\stackrel{\text{def}}{=} \#C > 0 \rightarrow\\
&\qquad (isRaReq(last(C)) \ \wedge\ isUniquePID(last(C).spid))\\
C^r_{ra} &\stackrel{\text{def}}{=} (\#C = \#D > 0) \rightarrow\\
&\qquad (isRaRes(last(D)) \ \wedge\ isValidCap(last(D).cap))
\end{aligned}$$

The assertion network for R_{ra} is as follows:

$$\begin{aligned}
Q_{s^r_{ra}} &\stackrel{\text{def}}{=} \#C = \#D = 0 \ \wedge\ |T_r| = 1\\
Q_{r_1} &\stackrel{\text{def}}{=} \#C = \#D\\
Q_{r_2} &\stackrel{\text{def}}{=} \#D = (\#C - 1) \ \wedge\ last(C) = x_r \ \wedge\ isRaReq(last(C)) \wedge\\
&\qquad isUniquePID(last(C).spid) \ \wedge\ isValidCap(rCap)\\
Q_{r_3} &\stackrel{\text{def}}{=} \#D = (\#C - 1) \ \wedge\ last(C) = x_r \wedge\\
&\qquad isRaRes(y_r) \ \wedge\ isValidCap(y_r.cap)\\
Q_{t^r_{ra}} &\stackrel{\text{def}}{=} false
\end{aligned}$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption A^r_{ra}, commitment C^r_{ra}, and channels C and D.

- $\models Q_{s^r_{ra}} \rightarrow C^r_{ra}$ follows from the above definitions.
- $\models Q_{s^r_{ra}} \wedge A^r_{ra} \rightarrow Q_{l_{r,1}} \circ init^r_{ra}$. In this internal transition, the size of the capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{r,1}} \ \wedge\ A^r_{ra} \rightarrow ((A^r_{ra} \rightarrow Q_{l_{r,2}}) \ \wedge\ C^r_{ra}) \circ (f_{r,1} \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_r,\ h \mapsto v,\ \sigma(h).(C, v))$ for arbitrary v. In this input transition, a communication through channel C just took place. Function g assigns the last received value from channel C to x_r. Since $\#C > 0$, the first implication of A^r_{ra} satisfies $isRaReq(last(C))$ and $isUniquePID(last(C).spid)$ predicates. Function $f_{r,1}$ allocates the requested resource and assigns the returned valid r-cap to $rCap$. Hence, Q_{r_2} satisfies $isValidCap(rCap)$ predicate. C^r_{ra} holds since $\#C \neq \#D$. Thus, this verification holds.

- $\models Q_{l_{r,2}} \wedge A^r_{ra} \rightarrow Q_{l_{r,3}} \circ (f_{r,3} \circ f_{r,2})$. In this internal transition, function $f_{i,2}$ inserts $rCap$ into T_r. Hence, $\sigma' \models |T_r| > 1$. Furthermore, function $f_{i,2}$ returns a valid i-cap, which points to the inserted resource capability inside the tree. Variable $iCap$ keeps the returned intermediate capability. Because $iCap$ points to the inserted $rCap$ inside T_r and $\sigma \models isValidCap(rCap)$, we have $\sigma' \models isValidCap(iCap)$. Function $f_{i,3}$ creates a resource-allocation response for program I_{ra} and assigns it to y_r, such that $isRaRes(y_r) = true$ and $y_r.cap = iCap$. In addition, because $isValidCap(iCap)$, we have $isValidCap(y_r.cap)$. Thus, this verification holds.

- $\models Q_{l_{r,3}} \wedge A^r_{ra} \rightarrow ((A^r_{ra} \rightarrow Q_{l_{r,1}}) \wedge C^r_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y_r)))$. In this output transition, program R_{ra} just sends y_r through channel C toward program I_{ra}. C^r_{ra} holds because $\#C = \#D > 0$, $last(D) = \sigma(y_r)$, and $\sigma \models isRaRes(y_r) \wedge isValidCap(y_i.cap)$. Hence, this verification holds.

  Since $Q_{t^r_{ra}} \rightarrow \psi^r_{ra}$, the post-condition of program R_{ra} is satisfied.

5.4.3 Resource Allocation - Parallel Composition

This section presents the parallel composition of the resource-allocation components based on the described rule in **??**. Consider the parallel composition $R_{ra} \parallel I_{ra}$. By applying the parallel composition rule, we deduce the following A-C formula:

$$\begin{gathered} \vdash \langle A_{ra},\ C_{ra} \rangle \\ \{\#A = \#B = \#C = \#D = 0 \ \wedge\ |T_r| = |T_i| = 1\} \\ R_{ra} \parallel I_{ra}\ \{false\} \end{gathered}$$

where A_{ra} and C_{ra} are as follows:

$$\begin{aligned} A_{ra} &\stackrel{\text{def}}{=} \#A > 0 \rightarrow \\ &\qquad isRaReq(last(A)) \ \wedge\ isUniquePID(last(A).spid)) \\ C_{ra} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D > 0 \rightarrow \\ &\qquad (isRaRes(last(B)) \ \wedge\ isValidCap(last(B).cap)) \end{aligned}$$

Because $\models A_{ra} \ \wedge\ C^i_{ra} \rightarrow A^r_{ra}$ and $\models A_{ra} \ \wedge\ C^r_{ra} \rightarrow A^i_{ra}$. Thus, $R_{ra} \parallel I_{ra}$ generates a valid capability when it receives a resource-allocation request from a process.

□

5.5 Second Lemma: Internal Delegation

The second lemma indicates that the internal delegation of a valid p-cap by a process creates a valid p-cap for another process inside the same node. Hence, the intermediate controller handles the internal delegation inside the node. We formally define the lemma and proof it.

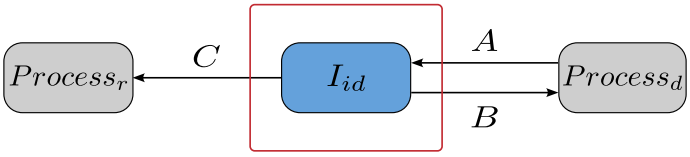


Figure 9: Syntactic Interfaces of the internal-delegation Programs.

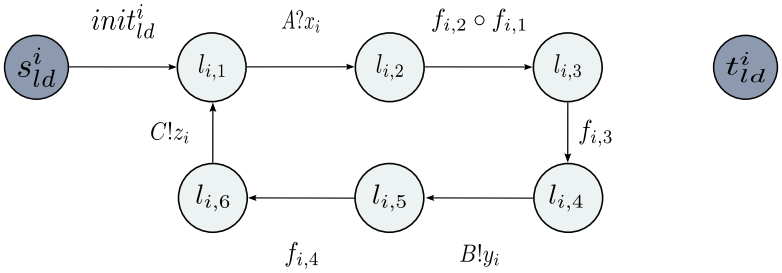


Figure 10: Internal Delegation - Program I_{id}.

Lemma 2. *Given two processes with unique IDs inside the same process node, delegation of a valid capability from the first process, $process_d$, to the second process, $process_r$, causes $process_r$ and $process_d$ to receive a valid process capability and a valid indicator capability, respectively.*

Proof. We use the assumption-commitment technique to prove the lemma.

We model the intermediate controller in a internal-delegation operation by program $I_{id} \stackrel{\text{def}}{=} (L_{i_id}, T_{i_id}, s_{i_id}, t_{i_id})$ to prove the lemma. Figure 9 depicts the syntactic interfaces between the program and two processes. In addition, figure 10 illustrates sequential diagrams of programs I_{id}. In the next step, we prove that $process_r$ receives a valid p-cap when $process_d$ delegates a valid capability. In addition, we prove that $process_d$ receives a valid p-cap of type indicator to the new capability inside the intermediate-capability tree.

5.5.1 internal Delegation - Intermediate Controller

Program I_{id} receives a internal capability-delegation request from $process_d$ via channel A. The request contains a valid p-cap that points to an intermediate capability inside the intermediate-capability tree. In the next step, program I_{id} delegates the intermediate capability according to the request and inserts the new intermediate capability inside the tree. Finally, program I_{id} creates a delegation response for $procces_r$ and a delegation-confirmation response for $process_d$, respectively. The delegation response and the delegation-confirmation response contain the new p-cap and the new indicator capability, respectively. The functions of I_{id} are as follows:

$$
\begin{aligned}
init^i_{id}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))),\\
f_{i,1}(\sigma) &= (\sigma : indCap \mapsto delegate(\sigma(rootPtr), \sigma(x_i))),
\end{aligned}
$$

$$
\begin{aligned}
f_{i,2}(\sigma) &= (\sigma : pCap \mapsto genCapFromInd(\sigma(icTree), \sigma(indCap))),\\
f_{i,3}(\sigma) &= (\sigma : y_i \mapsto genLdConfRes(\sigma(x_i), \sigma(indCap))),\\
f_{i,4}(\sigma) &= (\sigma : z_i \mapsto genLdRes(\sigma(x_i), \sigma(pCap)))
\end{aligned}
$$

The A-C formula for process I_{id} is as follows:

$$\vdash \langle Ass^i_{id},\ C^i_{id}\rangle\ \{\varphi^i_{id}\}\ I_{id}\ \{\psi^i_{id}\}$$

where the formal definition of φ^i_{id}, ψ^i_{id}, Ass^i_{id}, and C^i_{id} are as follows:

$$
\begin{aligned}
\varphi^i_{id} &\stackrel{\text{def}}{=} \#A = \#B = \#C = 0\ \wedge\ |icTree| > 1\\
\psi^i_{id} &\stackrel{\text{def}}{=} false\\
Ass^i_{id} &\stackrel{\text{def}}{=} \#A > 0 \rightarrow\\
&\quad (isIdReq(last(A))\ \wedge\ isValidCap(last(A).cap) \wedge\\
&\quad isCurrentNode(last(A).snid) \wedge\\
&\quad isCurrentNode(last(A).rnid)) \wedge\\
&\quad isUniquePID(last(A).spid)\ \wedge\ isUniquePID(last(A).rpid)))\\
C^i_{id} &\stackrel{\text{def}}{=} (\#A = \#B > \#C \rightarrow\\
&\quad isDelConfRes(last(B))\ \wedge\ isValidInd(last(B).cap)) \wedge\\
&\quad (\#A = \#B = \#C > 0 \rightarrow\\
&\quad isLdRes(last(C))\ \wedge\ isValidCap(last(C).cap)
\end{aligned}
$$

The assertion network for I_{id} is as follows:

$$
\begin{aligned}
Q_{s^i_{id}} &\stackrel{\text{def}}{=} \#A = \#B = \#C = 0\ \wedge\ |icTree| > 1\\
Q_{l_{i,1}} &\stackrel{\text{def}}{=} \#A = \#B = \#C\ \wedge\ |icTree| > 1\\
Q_{l_{i,2}} &\stackrel{\text{def}}{=} \#B = \#C = (\#A - 1)\ \wedge\ last(A) = x_i\ \wedge\ |icTree| > 1 \wedge\\
&\quad isIdReq(last(A))\ \wedge\ isValidCap(last(A).cap) \wedge\\
&\quad isCurrentNode(last(A).snid)\ \wedge\ isCurrentNode(last(A).rnid)) \wedge\\
&\quad isUniquePID(last(A).spid)\ \wedge\ isUniquePID(last(A).rpid)))\\
Q_{l_{i,3}} &\stackrel{\text{def}}{=} \#B = \#C = (\#A - 1)\ \wedge\ last(A) = x_i\ \wedge\ |icTree| > 2 \wedge\\
&\quad isValidInd(iCap))\ \wedge\ isValidCap(pCap))\\
Q_{l_{i,4}} &\stackrel{\text{def}}{=} \#B = \#C = (\#A - 1)\ \wedge\ last(A) = x_i\ \wedge\ |icTree| > 2 \wedge\\
&\quad isValidCap(pCap))\ \wedge\ isDelConfRes(y_i)\ \wedge\ isValidInd(y_i.cap)\\
Q_{l_{i,5}} &\stackrel{\text{def}}{=} \#A = \#B = (\#C + 1)\ \wedge\ last(A) = x_i\ \wedge\ \ last(B) = y_i \wedge\\
&\quad |icTree| > 2\ \wedge\ isLdRes(z_i)\ \wedge\ isValidCap(z_i.cap))\\
Q_{l_{i,6}} &\stackrel{\text{def}}{=} \#A = \#B = (\#C + 1)\ \wedge\ last(A) = x_i\ \wedge\ \ last(B) = y_i \wedge
\end{aligned}
$$

$$|icTree| > 2 \ \wedge \ isLdRes(z_i) \ \wedge \ isValidCap(z_i.cap))$$

$$Q_{t^i_{id}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^i_{id}, commitment C^i_{id}, and channels A, B, and C.

- $\models Q_{s^i_{id}} \rightarrow C^i_{id}$ follows from the above definitions.
- $\models Q_{s^i_{id}} \wedge Ass^i_{id} \rightarrow Q_{l_{i,1}} \circ init^i_{id}$. In this internal transition, the size of the intermediate-capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{i,1}} \ \wedge \ Ass^i_{id} \rightarrow ((Ass^i_{id} \rightarrow Q_{l_{i,2}}) \ \wedge \ C^i_{id}) \circ (f_{i,2} \circ f_{i,1} \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, \ h \mapsto v, \ \sigma(h).(A, v))$ for arbitrary value v. In this input transition, communication through channel A just took place, and function g assigns the received value to x_i. Since $\#A > 0$, the first implication of Ass^i_{id} satisfies that the received message is a internal-delegation request, and both the delegator and the receiver processes have unique PIDs. Thus, this verification holds.
- $\models Q_{l_{i,2}} \ \wedge \ Ass^i_{id} \rightarrow Q_{l_{i,3}} \circ (f_{i,2} \circ f_{i,1})$. In this internal transition, function $f_{i,1}$ delegates the $x_i.cap$, creates a valid indicator capability pointing to the new i-cap inside the intermediate-capability tree, and assigns it to $indCap$. Hence, we have $\sigma' \models |icTree| > 2$. Furthermore, function $f_{i,2}$ creates a valid p-cap corresponding to $indCap$ and assigns it to $pCap$. Hence, this verification holds.
- $\models Q_{l_{i,3}} \ \wedge \ Ass^i_{id} \rightarrow Q_{l_{i,4}} \circ f_{i,3}$. In this internal transition, function $f_{i,3}$ creates a delegation-confirmation response for the delegator process and assigns it to y_i, wherein $y_i.cap = indCap$. Hence, y_i contains a valid indicator capability. Thus, this verification holds.
- $\models Q_{l_{i,4}} \wedge Ass^i_{id} \rightarrow ((Ass^i_{id} \rightarrow Q_{l_{i,5}}) \wedge C^i_{id}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y_i)))$. In this output transition, program I_{id} sends y_i through channel B. C^i_{id} holds because $last(B) = \sigma(y_i)$ and $\sigma \models isLdConfRes(y_i) \ \wedge \ isValidInd(y_i.cap)$. Hence, this verification holds.
- $\models Q_{l_{i,5}} \wedge Ass^i_{id} \rightarrow Q_{l_{i,6}} \circ f_{i,4}$. In this internal transition, function $f_{i,4}$ creates a delegation response for the receiver process and assigns it to z_i, wherein $z_i.cap = pCap$. Hence, z_i contains a valid process capability. Thus, this verification holds.
- $\models Q_{l_{i,6}} \wedge Ass^i_{id} \rightarrow ((Ass^i_{id} \rightarrow Q_{l_{i,1}}) \wedge C^i_{id}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(z_i)))$. In this output transition, program I_{id} sends z_i through channel C. C^i_{id} holds because $\sigma \models isLdRes(z_i) \ \wedge \ isValidCap(z_i.cap)$. Hence, this verification holds.

  Since $Q_{t^i_{id}} \rightarrow \psi^i_{id}$, the post-condition of program I_{id} is satisfied.

□

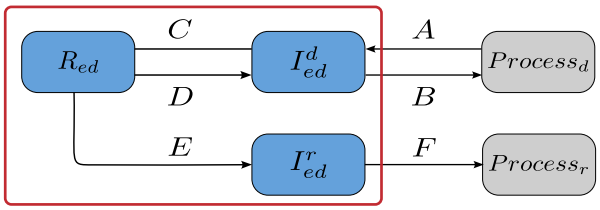


Figure 11: Syntactic Interfaces of the External-delegation Programs.

5.6 Third Lemma: External Delegation

The Third lemma indicates that delegating a valid p-cap by the delegator process creates a valid p-cap for a process inside another node. This operation requires that both intermediate controllers and the resource controller participate. We formally define the lemma and proof it.

Lemma 3. *Given two processes with unique PIDs inside different process nodes, delegating a valid capability from the first process to the second process causes that the receiver and the delegator processes receive a valid p-cap and a valid indicator, respectively.*

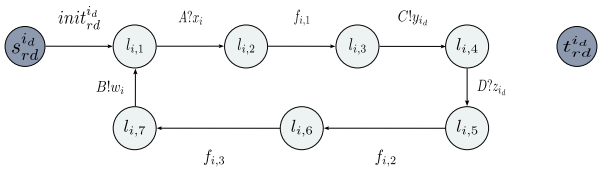
Proof. We use the assumption-commitment technique to prove the lemma.

To proof the lemma, we model the intermediate controller, operating at the delegator-process side, the resource controller, and the intermediate controller, operating at the receiver-process side, in a external-delegation operation by programs $I^d_{ed} \stackrel{\text{def}}{=} (L^{i_d}_{ed}, T^{i_d}_{ed}, s^{i_d}_{ed}, t^{i_d}_{ed})$, $R_{ed} \stackrel{\text{def}}{=} (L^r_{ed}, T^r_{ed}, s^r_{ed}, t^r_{ed})$, and $I^r_{ed} \stackrel{\text{def}}{=} (L^{i_r}_{ed}, T^{i_r}_{ed}, s^{i_r}_{ed}, t^{i_r}_{ed})$, respectively. Figure 11 depicts these three programs and their syntactic interfaces. The messages are exchanged between the programs via the channels to handle external-delegation requests. In addition, figures 12a, 12b, and 12c illustrate sequential diagrams of programs I^d_{ed}, R_{ed}, and I^r_{ed}, respectively. In the next step, we prove that the receiver process will receive a valid p-cap. In addition, we prove that process the delegator process receives a valid indicator capability.

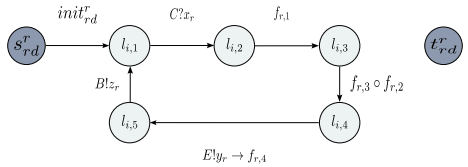
5.6.1 External Delegation - Delegating-Side Intermediate Controller

Program I^d_{ed} receives a external-delegation request from the process via channel A, creates a corresponding request for program R_{ed}, and sends it through channel C. Furthermore, It receives the response from program R_{ed} via channel D. Program I^d_{ed} updates its capability tree when it receives an i-cap from the resource controller by inserting the indicator capability inside its tree. In addition, it creates the corresponding indicator capability for the inserted capability inside its tree. Finally, program I^d_{ed} creates a response for the process and sends the indicator inside the response to the process through channel B. The conditions and functions of I^d_{ed} are as follow:

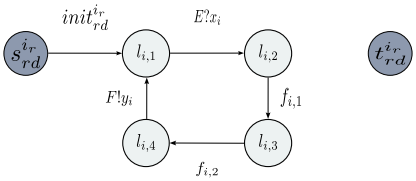
$$
\begin{aligned}
init^{i_d}_{ed}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))),\\
f_{i,1}(\sigma) &= (\sigma : y_i \mapsto genRdReq(\sigma(rootPtr), \sigma(x_i)),
\end{aligned}
$$



(a) Program I^d_{ed}.



(b) Program R_{ed}.



(c) Program I^r_{ed}.

Figure 12: external-Delegation Processes.

$$
\begin{aligned}
f_{i,2}(\sigma) &= (\sigma : pIndCap \mapsto insert(\sigma(icTree), \sigma(z_i))),\\
f_{i_d,3}(\sigma) &= (\sigma : w_i \mapsto genRdConfRes(\sigma(x), \sigma(pIndCap)))
\end{aligned}
$$

The A-C formula for process I^d_{ed} is as follows:

$$\vdash \langle Ass^{i_d}_{ed},\ C^{i_d}_{ed}\rangle\ \{\varphi^{i_d}_{ed}\}\ I^d_{ed}\ \{\psi^{i_d}_{ed}\}$$

where the formal definition of $\varphi^{i_d}_{ed}$, $\psi^{i_d}_{ed}$, $Ass^{i_d}_{ed}$, and $C^{i_d}_{ed}$ are as follows:

$$
\begin{aligned}
\varphi^{i_d}_{ed} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0\ \wedge\ |icTree| > 1\\
\psi^{i_d}_{ed} &\stackrel{\text{def}}{=} false\\
Ass^{i_d}_{ed} &\stackrel{\text{def}}{=} (\#A > 0 \rightarrow\\
&\quad (isRdReq(last(A))\ \wedge\ isValidCap(last(A).cap) \wedge\\
&\quad isUniquePID(last(A).spid) \wedge\\
&\quad isUniquePID(last(A.rpid))) \wedge\\
&\quad (\#C = \#D > 0 \rightarrow\\
&\qquad isEdConfRes(last(D))\ \wedge\ isValidInd(last(D).cap))) \wedge\\
C^{i_d}_{ed} &\stackrel{\text{def}}{=} (\#A = \#B > 0 \rightarrow
\end{aligned}
$$

$$isEdConfRes(last(B)) \ \wedge \ isValidInd(last(B).cap)) \wedge$$
$$(\#C > 0 \rightarrow (isRdReq(last(C)) \ \wedge \ isValidCap(last(C).cap) \wedge$$
$$isUniquePID(last(C).spid) \ \wedge \ isUniquePID(last(C).rpid)))$$

The assertion network for I^d_{ed} is as follows:

$$Q_{s^{i_d}_{ed}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree| > 1$$

$$Q_{l_{i,1}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D$$

$$Q_{l_{i,2}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \wedge$$
$$isRdReq(last(A)) \ \wedge \ isValidCap(last(A).cap)) \wedge$$
$$isUniquePID(last(A).spid) \ \wedge \ isUniquePID(last(A).rpid)$$

$$Q_{l_{i,3}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \wedge$$
$$isRdReq(y_i) \ \wedge \ isValidInd(y_i.cap)) \ \wedge \ isUniquePID(y_i.spid) \wedge$$
$$isUniquePID(y_i.rpid)$$

$$Q_{l_{i,4}} \stackrel{\text{def}}{=} \#A = \#B = (\#A - 1) = (\#C - 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i$$

$$Q_{l_{i,5}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \wedge$$
$$last(D) = z_i \ \wedge \ isEdConfRes(last(D)) \ \wedge \ isValidInd(last(D).cap)$$

$$Q_{l_{i,6}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \wedge$$
$$last(D) = z_i \ \wedge \ isValidInd(pIndCap)$$

$$Q_{l_{i,7}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \wedge$$
$$last(D) = z_i \ \wedge \ isEdConfRes(w_i) \ \wedge \ isValidInd(w_i.cap)$$

$$Q_{t^{i_d}_{ed}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption $Ass^{i_d}_{ed}$, commitment $C^{i_d}_{ed}$, and channels A, B, C, and D.

- $\models Q_{s^{i_d}_{ed}} \rightarrow C^{i_d}_{ed}$ follows from the above definitions.
- $\models Q_{s^{i_d}_{ed}} \ \wedge \ Ass^{i_d}_{ed} \rightarrow Q_{l_{i,1}} \circ init^{i_d}_{ed}$. In this internal transition, just function $init^{i^d}_{ed}$ gets a pointer to the root of the capability tree. Hence, this verification holds.
- $\models Q_{l_{i,1}} \wedge Ass^{i_d}_{ed} \rightarrow ((Ass^{i_d}_{ed} \rightarrow Q_{l_{i,2}}) \wedge C^{i_d}_{ed}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, h \mapsto v, \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Because $\#A > 0$, the first implication of $Ass^{i_d}_{ed}$ satisfies that it is a external-delegation request, $last(A).cap$ is a valid indicator capability, and the delegator and receiver PIDs are unique. Thus, this verification holds.

- $\models Q_{l_{i,2}} \wedge Ass^{i_d}_{ed} \rightarrow Q_{l_{i,3}} \circ f_{i,1}$. In this internal transition, function $f_{i,1}$ creates a external-revocation request and assigns it to variable y_i, such that $isRdReq(y_i) = true$, $y_i.spid = last(A).spid$, and $y_i.rpid = last(A).rpid$. Hence, the delegator and receiver PIDs in the new request are unique because Ass^i_{ra} indicates that both PIDs inside the request from the process are unique. Thus, this verification holds.

- $\models Q_{l_{i,3}} \wedge Ass^{i_d}_{ed} \rightarrow ((Ass^{i_d}_{ed} \rightarrow Q_{l_{i,7}}) \wedge C^{i_d}_{ed}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y_i)))$. In this output transition, program I^d_{ed} sends y_i through channel C. $C^{i_d}_{ed}$ holds because $last(C) = \sigma(y_i)$, $isRdReq(y_i)$, and both $y_i.spid$ and $y_i.rpid$ are unique. Hence, this verification holds.

- $\models Q_{l_{i,4}} \wedge Ass^{i_d}_{ed} \rightarrow ((Ass^{i_d}_{ed} \rightarrow Q_{l_{i,5}}) \wedge C^{i_d}_{ed}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z_i,\ h \mapsto v,\ \sigma(h).(D, v))$ for arbitrary value v. In this input transition, a communication through channel D took place, and function g assigns the received value to z_i. Because $\#D > 0$, $Ass^{i_d}_{ed}$ implies that $isEdConfRes(last(D))$ and $isValidInd(last(D).cap)$. Thus, this verification holds.

- $\models Q_{l_{i,5}} \wedge Ass^{i_d}_{ed} \rightarrow Q_{l_{i,6}} \circ f_{i,2}$. In this internal transition, function $f_{i,2}$ inserts the received intermediate indicator into the capability tree and returns $pIndCap$ that points to it inside the tree. We have $isValidCap(pIndCap)$ because $ValidInd(last(D).cap)$, and $pIndCap$ points to the it after the intermediate controller inserts it inside the tree. Thus, this verification holds.

- $\models Q_{l_{i,6}} \wedge Ass^{i_d}_{ed} \rightarrow Q_{l_{i,7}} \circ (f_{i,3} \circ f_{i,2})$. In this internal transition, function $f_{i,3}$ creates a external-delegation confirmation response for the process and assigns it to w_i, wherein $w_i.cap = pIndCap$. Thus, this verification holds.

- $\models Q_{l_{i,7}} \wedge Ass^{i_d}_{ed} \rightarrow ((Ass^{i_d}_{ed} \rightarrow Q_{l_{i,1}}) \wedge C^{i_d}_{ed}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(w_i)))$. In this output transition, program I^d_{ed} sends w_i through channel B. $C^{i_d}_{ed}$ holds because $\sigma \models isEdConfRes(w_i) \wedge isValidInd(w_i.cap)$. Hence, this verification holds.

  Because $Q_{t^{i_d}_{ed}} \rightarrow \psi^{i_d}_{ed}$, the post-condition of program I^d_{ed} is satisfied.

5.6.2 External Delegation - Resource Controller

Program R_{ed} receives a external-delegation request from program I^d_{ed} via channel C. In the next step, program R_{ed} creates creates the delegated capability and inserts it inside the capability tree. Furthermore, it creates the delegated resource capability's corresponding intermediate and indicator capabilities. Afterward, program R_{ed} creates the external-delegation and confirmation responses and inserts the intermediate and indicator capabilities inside them. Finally, it sends the responses through channels E and D toward programs I^r_{ed} and I^d_{ed}, respectively. The functions of R_{ed} are as follows:

$$
\begin{aligned}
init^r_{ed}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(rcTree))),\\
f_{r,1}(\sigma) &= (\sigma : indCap \mapsto delegate(\sigma(rootPtr), \sigma(x_r))),\\
f_{r,2}(\sigma) &= (\sigma : iCap \mapsto genCapFromInd(\sigma(indCap), \sigma(indCap))),
\end{aligned}
$$

$$f_{r,3}(\sigma) = (\sigma : y_r \mapsto genRdRes(\sigma(x_r), \sigma(iCap))),$$
$$f_{r,3}(\sigma) = (\sigma : z_r \mapsto genRdConfRes(\sigma(x_r), \sigma(indCap))),$$

where $rcTree$ is the resource-capability tree, $indCap$ is an intermediate indicator, and $iCap$ is an intermediate capability.

The A-C formula for process R_{ed} is as follows:

$$\vdash \langle Ass^r_{ed}\ C^r_{ed} \rangle\ \{\varphi^r_{ed}\}\ R_{ed}\ \{\psi^r_{ed}\}$$

where the formal definition of φ_r, ψ_r , Ass_r, and C_r are as follows:

$$\begin{aligned}
\varphi^r_{ed} &\stackrel{\text{def}}{=} \#C = \#D = \#E = 0 \ \wedge\ |rcTree| \geqslant 1 \\
\psi^r_{ed} &\stackrel{\text{def}}{=} false \\
Ass^r_{ed} &\stackrel{\text{def}}{=} (\#C > 0 \rightarrow \\
&\quad (isRdReq(last(C)) \ \wedge\ isValidInd(last(C).cap) \wedge \\
&\quad isUniquePID(last(C).spid) \wedge \\
&\quad isUniquePID(last(C).rpid))) \\
C^r_{ed} &\stackrel{\text{def}}{=} (\#C = \#D > 0 \rightarrow \\
&\quad (isEdConfRes(last(D)) \ \wedge\ isValidInd(last(D).cap))) \wedge \\
&\quad (\#C = \#E > 0 \rightarrow \\
&\quad (isRdRes(last(E)) \ \wedge\ isValidCap(last(E).cap)))
\end{aligned}$$

The assertion network for R_{ed} is as follows:

$$\begin{aligned}
Q_{s^r_{ed}} &\stackrel{\text{def}}{=} \#C = \#D = \#E = 0 \ \wedge\ |rcTree| \geqslant 1 \\
Q_{r_1} &\stackrel{\text{def}}{=} \#C = \#D = \#E \\
Q_{r_2} &\stackrel{\text{def}}{=} \#D = \#E = (\#C - 1) \ \wedge\ last(C) = x_r \ \wedge\ isRdReq(last(C)) \wedge \\
&\quad isValidInd(last(C).cap) \ \wedge\ isUniquePID(last(C).spid) \wedge \\
&\quad isUniquePID(last(C).rpid) \\
Q_{r_3} &\stackrel{\text{def}}{=} \#D = \#E = (\#C - 1) \ \wedge\ last(C) = x_r \ \wedge\ isValidInd(indCap) \\
Q_{r_4} &\stackrel{\text{def}}{=} \#D = \#E = (\#C - 1) \ \wedge\ last(C) = x_r \ \wedge\ isValidInd(indCap) \wedge \\
&\quad isRdRes(y_r) \ \wedge\ isValidCap(y_r.cap) \\
Q_{r_5} &\stackrel{\text{def}}{=} \#C = \#E = (\#D + 1) \ \wedge\ last(C) = x_r \ \wedge\ last(E) = y_r \wedge \\
&\quad isEdConfRes(z_r) \ \wedge\ isValidInd(z_r.cap) \\
Q_{t^r_{ed}} &\stackrel{\text{def}}{=} false
\end{aligned}$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^r_{ed}, commitment C^r_{ed}, and channels C, D, and E.

- $\models Q_{s^r_{ed}} \rightarrow C^r_{ed}$ follows from the above definitions.

- $\models Q_{s^r_{ed}} \wedge Ass^r_{ed} \rightarrow Q_{l_{r,1}} \circ init^r_{ed}$. In this internal transition, just function $init^r_{ed}$ gets a pointer to the root of the capability tree. Hence, this verification holds.

- $\models Q_{l_{r,1}} \wedge Ass^r_{ed} \rightarrow ((Ass^r_{ed} \rightarrow Q_{r_2}) \wedge C^r_{ed}) \circ$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_r, h \mapsto v, \sigma(h).(C, v))$ for arbitrary v. In this input transition, a communication through channel C just took place. Function g assigns the last received value from channel C to x_r. Because $\#C > 0$, the first implication of Ass^r_{ed} satisfies that the request is a resource allocation one, the process IDs of the delagator and receiver processes are unique, delegator, and the request contains a valid indicator. C^r_{ed} holds because neither $\#C = \#D$ nor $\#C = \#E$. Thus, this verification holds.

- $\models Q_{l_{r,2}} \wedge Ass^r_{ed} \rightarrow Q_{l_{r,3}} \circ (unlockTree \circ f_{r,1})$. In this internal transition, function $f_{r,1}$ delegates the capability and inserts the newly generated capability into the capability tree. Furthermore, function $f_{r,1}$ returns a valid indicator capability that points to the inserted resource capability inside the tree. Variable $indCap$ keeps the returned intermediate indicator. Hence, we have $\sigma' \models isValidInd(indCap)$. Thus, this verification holds.

- $\models Q_{l_{r,3}} \wedge Ass^r_{ed} \rightarrow Q_{l_{r,4}} \circ (f_{r,3} \circ f_{r,2})$. In this internal transition, function $f_{r,2}$ generates a intermediate capability from the newly generated resource capability and assign it to variable $iCap$. Hence, we have $isValidCap(iCap)$. Afterward, function $f_{r,3}$ creates external-delegation response and inserts $iCap$ as its capability. Thus, this verification holds.

- $\models Q_{l_{r,4}} \wedge Ass^r_{ed} \rightarrow ((Ass^r_{ed} \rightarrow Q_{l_{r,5}}) \wedge C^r_{ed}) \circ (f_{r,3} \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(E, \sigma(y_r)))$. In this output transition, program R_{ed} sends y_r through channel E. Commitment C^r_{ed} holds because we have $isRdRes(y_r)$ and $isValidCap(y_r.cap)$. Afterward, function $f_{r,3}$ creates a external-delegation confirmation response, inserts $indCap$ as its capability, and assigns it to variable z_r. Hence, it satisfies both $isEdConfRes(z_r)$ and $isValidInd(z_r.cap)$. Thus, this verification holds.

- $\models Q_{l_{r,5}} \wedge Ass^r_{ed} \rightarrow ((Ass^r_{ed} \rightarrow Q_{l_{r,1}}) \wedge C^r_{ed}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y_r)))$. In this output transition, program R_{ed} sends z_r through channel B. Commitment C^r_{ed} holds because Z_r is a external-delegation-confirmation response and contains a valid indicator. Thus, this verification holds.

  Because $Q_{t^r_{ed}} \rightarrow \psi^r_{ed}$, the post-condition of program R_{ed} is satisfied.

5.6.3 External Delegation - Receiving-Side Intermediate Controller

Program I^r_{ed} receives a external-delegation request from the process via channel E. Then, it updates its capability tree by inserting the received intermediate capability inside the tree. In addition, it creates the corresponding p-cap for the inserted i-cap. Finally, program I^r_{ed} creates a external-delegation response for the receiving process

and sends the p-cap inside the response to the process through channel B. The functions of I^r_{ed} are as follows:

$$\begin{aligned} init^{i_r}_{ed}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))), \\ f_{i,1}(\sigma) &= (\sigma : pCap \mapsto insert(\sigma(rootPtr), \sigma(x_i.cap))) \\ f_{i,2}(\sigma) &= (\sigma : y_i \mapsto genRdRes(\sigma(x_i), \sigma(pCap))), \end{aligned}$$

The A-C formula for process I^d_{ed} is as follows:

$$\vdash \langle Ass^{i_r}_{ed},\ C^{i_r}_{ed} \rangle\ \{\varphi^{i_r}_{ed}\}\ I^r_{ed}\ \{\psi^{i_r}_{ed}\}$$

where the formal definition of $\varphi^{i_r}_{ed}$, $\psi^{i_r}_{ed}$, $Ass^{i_r}_{ed}$, and $C^{i_r}_{ed}$ are as follows:

$$\begin{aligned} \varphi^{i_r}_{ed} &\stackrel{\text{def}}{=} & \#E = \#F = 0\ \wedge\ |icTree| \geqslant 1 \\ \psi^{i_r}_{ed} &\stackrel{\text{def}}{=} & false \\ Ass^{i_r}_{ed} &\stackrel{\text{def}}{=} & \#E > 0 \rightarrow \\ & & (isRdRes(last(E))\ \wedge\ isValidCap(last(E).cap) \\ C^{i_r}_{ed} &\stackrel{\text{def}}{=} & \#E = \#F > 0 \rightarrow \\ & & isRdRes(last(F))\ \wedge\ isValidCap(last(F).cap) \end{aligned}$$

The assertion network for I^r_{ed} is as follows:

$$\begin{aligned} Q_{s^{i_r}_{ed}} &\stackrel{\text{def}}{=} \#E = \#F = 0\ \wedge\ |icTree| \geqslant 1 \\ Q_{l_{i,1}} &\stackrel{\text{def}}{=} \#E = \#F \\ Q_{l_{i,2}} &\stackrel{\text{def}}{=} \#F = (\#E - 1)\ \wedge\ last(E) = x_i \wedge \\ & \quad isRdRes(last(E))\ \wedge\ isValidCap(last(E).cap)) \\ Q_{l_{i,3}} &\stackrel{\text{def}}{=} \#F = (\#E - 1)\ \wedge\ last(E) = x_i \wedge \\ & \quad isValidCap(pCap) \\ Q_{l_{i,4}} &\stackrel{\text{def}}{=} \#F = (\#E - 1)\ \wedge\ last(E) = x_i \wedge \\ & \quad isRdRes(y_i)\ \wedge\ isValidCap(y_i.cap)) \\ Q_{t^{i_d}_{ed}} &\stackrel{\text{def}}{=} false \end{aligned}$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption $Ass^{i_r}_{ed}$, commitment $C^{i_r}_{ed}$, and channels E and F.

- $\models Q_{s^{i_r}_{ed}} \rightarrow C^{i_r}_{ed}$ follows from the above definitions.

- $\models Q_{s^{i^r}_{ed}} \wedge Ass^{i_r}_{ed} \rightarrow Q_{l_{i,1}} \circ init^{i^r}_{ed}$. In this internal transition, just function $init^{i^r}_{ed}$ gets a pointer to the root of the capability tree. Hence, this verification holds.

- $\models Q_{l_{i,1}} \wedge Ass^{i_r}_{ed} \rightarrow ((Ass^{i_r}_{ed} \rightarrow Q_{l_{i,2}}) \wedge C^{i_r}_{ed}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, h \mapsto v, \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Because $\#A > 0$, the first implication of $Ass^{i_r}_{ed}$ satisfies that the last received message is a external-delegation response,and the capability inside it is a valid intermediate capability. Thus, this verification holds.

- $\models Q_{l_{i,2}} \wedge Ass^{i_r}_{ed} rightarrow Q_{l_{i,3}} \circ f_{i,1}$. In this internal transition, function $f_{i,1}$ inserts the received intermediate capability into the capability tree and returns $pCap$ that points to it inside the tree. Hence, we have $isValidCap(pCap)$. Thus, this verification holds.

- $\models Q_{l_{i,3}} \wedge Ass^{i_r}_{ed} \rightarrow Q_{l_{i,4}} \circ (f_{i,2} \circ unlockTree)$. In this internal transition, function $f_{i,2}$ creates a external-delegation response for the process and assigns it to y_i, wherein $y_i.cap = pCap$. Hence, this verification holds.

- $\models Q_{l_{i,4}} \wedge Ass^{i_r}_{ed} \rightarrow ((Ass^{i_r}_{ed} \rightarrow Q_{l_{i,1}}) \wedge C^{i_r}_{ed}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(F, \sigma(y_i)))$. In this output transition, program I^r_{ed} sends y_i through channel F. $C^{i_r}_{ed}$ holds because $\sigma \models isRdRes(y_i) \wedge isValidCap(y_i.cap)$. Thus, this verification holds.

  Because $Q_{t^{i_r}_{ed}} \rightarrow \psi^{i_r}_{ed}$, the post-condition of program I^r_{ed} is satisfied.

5.6.4 External Delegation - Parallel Composition

This section presents the parallel composition of the external-delegation components based on the described rule in 1. Consider the parallel composition $R_{ed} \parallel I^d_{ed} \parallel I^r_{ed}$. Program R_{ed} satisfies the assumption-commitment pair $(Ass^r_{ed},\ C^r_{ed})$ as follows:

$$\vdash \langle Ass^r_{ed}\ C^r_{ed} \rangle\ \{\varphi^r_{ed}\}\ R_{ed}\ \{\psi^r_{ed}\}$$

where the formal definition of φ_r, ψ_r , Ass_r, and C_r are as follows:

$$\begin{aligned}
\varphi^r_{ed} &\stackrel{\text{def}}{=} \#C = \#D = \#E = 0 \ \wedge\ |rcTree| \geqslant 1 \\
\psi^r_{ed} &\stackrel{\text{def}}{=} false \\
Ass^r_{ed} &\stackrel{\text{def}}{=} (\#C > 0 \rightarrow \\
&\quad (isRdReq(last(C)) \ \wedge\ isValidInd(last(C).cap) \wedge \\
&\quad\ isUniquePID(last(C).spid) \wedge \\
&\quad\ isUniquePID(last(C).rpid))) \\
C^r_{ed} &\stackrel{\text{def}}{=} (\#C = \#D > 0 \rightarrow \\
&\quad (isEdConfRes(last(D)) \ \wedge\ isValidInd(last(D).cap))) \wedge \\
&\quad (\#C = \#E > 0 \rightarrow
\end{aligned}$$

$$(isRdRes(last(E)) \ \wedge \ isValidCap(last(E).cap)))$$

Program I^d_{ed} satisfies the assumption-commitment pair $(Ass^{i_d}_{ed},\ C^{i_d}_{ed})$ as follows:

$$\vdash \langle Ass^{i_d}_{ed},\ C^{i_d}_{ed}\rangle\ \{\varphi^{i_d}_{ed}\}\ I^d_{ed}\ \{\psi^{i_d}_{ed}\}$$

where the formal definition of $\varphi^{i_d}_{ed}$, $\psi^{i_d}_{ed}$, $Ass^{i_d}_{ed}$, and $C^{i_d}_{ed}$ are as follows:

$$\begin{array}{lll}
\varphi^{i_d}_{ed} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree_d| > 1 \\
\psi^{i_d}_{ed} & \stackrel{\text{def}}{=} & false \\
Ass^{i_d}_{ed} & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow \\
 & & (isRdReq(last(A)) \ \wedge \ isValidCap(last(A).cap) \wedge \\
 & & isUniquePID(last(A).spid) \wedge \\
 & & isUniquePID(last(A.rpid))) \wedge \\
 & & (\#C = \#D > 0 \rightarrow \\
 & & \quad isEdConfRes(last(D)) \ \wedge \ isValidInd(last(D).cap))) \wedge \\
C^{i_d}_{ed} & \stackrel{\text{def}}{=} & (\#A = \#B > 0 \rightarrow \\
 & & \quad isEdConfRes(last(B)) \ \wedge \ isValidInd(last(B).cap)) \wedge \\
 & & (\#C > 0 \rightarrow \\
 & & (isRdReq(last(C)) \ \wedge \ isValidCap(last(C).cap) \wedge \\
 & & isUniquePID(last(C).spid) \ \wedge \ isUniquePID(last(C).rpid)))
\end{array}$$

Program I^r_{ed} satisfies the assumption-commitment pair $(Ass^{i_r}_{ed},\ C^{i_r}_{ed})$ as follows:

$$\vdash \langle Ass^{i_r}_{ed},\ C^{i_r}_{ed}\rangle\ \{\varphi^{i_r}_{ed}\}\ I^r_{ed}\ \{\psi^{i_r}_{ed}\}$$

where the formal definition of $\varphi^{i_r}_{ed}$, $\psi^{i_r}_{ed}$, $Ass^{i_r}_{ed}$, and $C^{i_r}_{ed}$ are as follows:

$$\begin{array}{lll}
\varphi^{i_r}_{ed} & \stackrel{\text{def}}{=} & \#E = \#F = 0 \ \wedge \ |icTree_r| \geqslant 1 \\
\psi^{i_r}_{ed} & \stackrel{\text{def}}{=} & false \\
Ass^{i_r}_{ed} & \stackrel{\text{def}}{=} & \#E > 0 \rightarrow \\
 & & (isRdRes(last(E)) \ \wedge \ isValidCap(last(E).cap) \\
C^{i_r}_{ed} & \stackrel{\text{def}}{=} & \#E = \#F > 0 \rightarrow \\
 & & \quad isRdRes(last(F)) \ \wedge \ isValidCap(last(F).cap)
\end{array}$$

By applying the parallel composition rule, we deduce as follows:

$$\begin{array}{c}
\vdash \langle Ass_{ed},\ C_{ed}\rangle \\
\{\varphi_{ed}\}\ R_{ra} \parallel I^d_{ed} \parallel I^r_{ed}\ \{\psi_{ed}\}
\end{array}$$

where the formal definition of φ_{ed}, ψ_{ed}, Ass_{ed}, and C_{ed} are as follows:

$$
\begin{aligned}
\varphi_{ed} \quad &\stackrel{\text{def}}{=} \quad \#A = \#B = \#C = \#D = \#E = \#F = 0 \ \wedge \ |rcTree| \geqslant 1 \ \wedge \\
&\qquad |icTree_d| \geqslant 1 \ \wedge \ |icTree_r| \geqslant 1 \\
\psi_{ed} \quad &\stackrel{\text{def}}{=} \quad false \\
Ass_{ed} \quad &\stackrel{\text{def}}{=} \quad \#A > 0 \rightarrow \\
&\qquad (isRdReq(last(A)) \ \wedge \ isValidCap(last(A).cap) \wedge \\
&\qquad isUniquePID(last(A).spid) \wedge \\
&\qquad isUniquePID(last(A).rpid)) \\
C_{ed} \quad &\stackrel{\text{def}}{=} \quad \#A = \#B = \#C = \#D = \#E = \#F > 0 \rightarrow \\
&\qquad (isEdConfRes(last(B)) \ \wedge \ isValidInd(last(B).cap) \\
&\qquad (isRdRes(last(F)) \ \wedge \ isValidCap(last(F).cap))
\end{aligned}
$$

□

5.7 Fourth Lemma: External Revocation

The fourth lemma is about the external-revocation operation. We first describe the external revocation instead of the internal revocation because a process may re-delegate a internally delegated capability to a external process. Hence, a internal-revocation operation may implicitly include several external-revocation operations. The fourth lemma indicates that the external revocation of a valid p-cap by the delegator process will invalidate the delegated p-cap owned by another process inside another node. Since it is a external-revocation operation, the intermediate controller in the delegator-process side and the resource controller handle the revocation together We formally define the lemma and proof it.

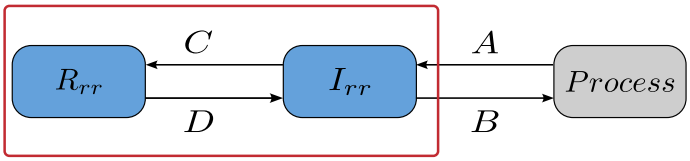
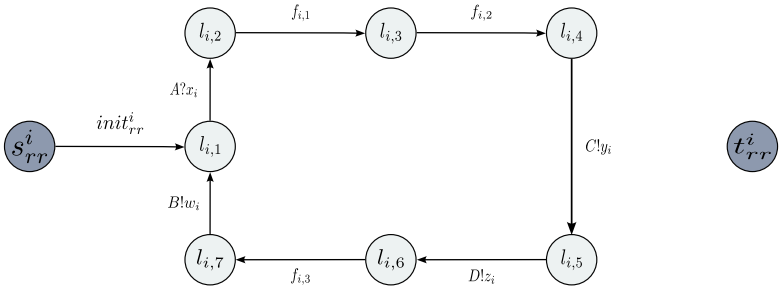


Figure 13: Syntactic Interfaces of the external-revocation Programs.

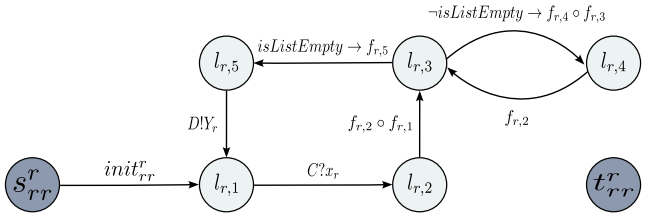
Lemma 4. *Given a process with a unique PID in a process node, an intermediate controller in the same node, and a resource controller in a resource node, revoking a delegated p-cap, which points to a resource inside the resource node, to another process inside another node invalidates the delegated p-cap.*

Proof. We use the assumption-commitment technique to prove the lemma.

We model the intermediate controller at the delegator-process side and the resource controller in a external-revocation operation by programs $I_{er} \stackrel{\text{def}}{=} (L^i_{er}, T^i_{er}, s^i_{er}, t^i_{er})$ and



(a) Program I_{er}.



(b) Program R_{er}.

Figure 14: external Revocation Programs.

$R_{er} \stackrel{\text{def}}{=} (L^r_{er}, T^r_{er}, s^r_{er}, t^r_{er})$, respectively. The programs and the process exchange messages via synchronous channels to handle external-revocation requests. Figure 13 depicts these two programs and their syntactic interfaces. Furthermore, figures 14a and 14b illustrate sequential diagrams of programs I_{er} and R_{er}, respectively. In the next step, we prove that when the process sends a external-revocation request, programs I_{er} and R_{er} revoke the delegated capability and all its re-delegated capabilities.

We need to define a loop-invariant for program I_{er} and a loop-invariant for program R_{er} to prove the correctness of the lemma. When an intermediate controller conducts a external-delegation operation, it inserts an indicator capability as the child of the delegated capability inside its capability tree. There would be no sub-tree after the inserted capability inside the tree because of its indicator type. On the other hand, the resource controller inserts a r-cap inside its capability tree. Hence, the capability may have a sub-tree because of the re-delegation(s).

5.7.1 External Revocation - Intermediate Controller

Program I_{er} receives a external-revocation request from the process via channel A. The request contains a valid process indicator that points to an intermediate indicator inside the intermediate-capability tree. In the next step, program I_{er} copies the intermediate indicator and removes it from the tree afterward. Furthermore, it creates a corresponding external-revocation request and sends it through channel C to program R_{er}. Finally, when program I_{er} receives the response from program R_{er} via channel D, it creates a response for the process and sends the response toward the process through channel B.

The functions of I_{er} are as follow:

$$\begin{aligned}
init^i_{er}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))),\\
f_{i,1}(\sigma) &= (\sigma : iIndCap \mapsto revoke(\sigma(rootPtr), \sigma(x_i.cap))),\\
f_{i,2}(\sigma) &= (\sigma : y_i \mapsto genRrReq(\sigma(x_i), \sigma(iIndCap))),\\
f_{i,3}(\sigma) &= (\sigma : w_i \mapsto genRevConfRes(\sigma(z_i)))
\end{aligned}$$

The A-C formula for program I_{er} is as follows:

$$\vdash \langle Ass^i_{er},\ C^i_{er}\rangle\ \{\varphi^i_{er}\}\ I_{er}\ \{\psi^i_{er}\}$$

where the formal definition of φ^i_{er}, ψ^i_{er}, Ass^i_{er}, and C^i_{er} are as follows:

$$\begin{aligned}
\varphi^i_{er} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0\ \wedge\ |icTree| \geqslant 1\\
\psi^i_{er} &\stackrel{\text{def}}{=} false\\
Ass^i_{er} &\stackrel{\text{def}}{=} (\#A > 0 \rightarrow\\
&\quad isRrReq(last(A))\ \wedge\ isUniquePID(last(A).spid)))\\
&\quad isValidInd(last(A).cap)) \wedge\\
&\quad (\#C = \#D > 0 \rightarrow isRrConfRes(last(D)))\\
C^i_{er} &\stackrel{\text{def}}{=} (\#A = \#B = \#C = \#D > 0 \rightarrow isRrConfRes(last(B))) \wedge\\
&\quad (\#C > 0 \rightarrow isRrConfRes(last(C)))
\end{aligned}$$

The assertion network for I_{er} is as follows:

$$\begin{aligned}
Q_{s^i_{er}} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0\ \wedge\ |icTree \geqslant 2\\
Q_{l_{i,1}} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D\ \wedge\ |icTree| \geqslant 2\\
Q_{l_{i,2}} &\stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1)\ \wedge\ last(A) = x_i\ \wedge\ |icTree| > 2 \wedge\\
&\quad isRrReq(last(A))\ \wedge\ isUniquePID(last(A).spid) \wedge\\
&\quad isValidInd(last(A).cap)\\
Q_{l_{i,3}} &\stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1)\ \wedge\ last(A) = x_i\ \wedge\ |icTree| \geqslant 2 \wedge\\
&\quad isValidInd(iIndCap)\ \wedge\ isTreeLocked\\
Q_{l_{i,5}} &\stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1)\ \wedge\ last(A) = x_i\ \wedge\ |icTree| \geqslant 2 \wedge\\
&\quad isRrReq(y_i)\ \wedge\ isUniquePID(y_i.spid)\ \wedge\ isValidInd(y_i.cap)\\
Q_{l_{i,6}} &\stackrel{\text{def}}{=} \#B = \#D = (\#A - 1) = (\#C - 1)\ \wedge\ last(A) = x_i\ \wedge\ last(C) = y_i \wedge\\
&\quad |icTree| \geqslant 2\\
Q_{l_{i,7}} &\stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1)\ \wedge\ last(A) = x_i\ \wedge\ last(C) = y_i \wedge\\
&\quad last(D) = z_i\ \wedge\ isRrConfRes(z_i)\ \wedge\ |icTree| \geqslant 2\\
Q_{l_{i,8}} &\stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1)\ \wedge\ last(A) = x_i\ \wedge\ last(C) = y_i \wedge
\end{aligned}$$

$$last(D) = z_i \ \wedge \ |icTree| \geqslant 2 \ \wedge \ isRrConfRes(w_i)$$

$$Q_{t^i_{er}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^i_{er}, commitment C^i_{er}, and channels A, B, C, and D.

- $\models Q_{s^i_{er}} \rightarrow C^i_{er}$ follows from the above definitions.
- $\models Q_{s^i_{er}} \wedge Ass^i_{er} \rightarrow Q_{l_{i,1}} \circ init^i_{er}$. In this internal transition, the size of the intermediate-capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{i,1}} \wedge Ass^i_{er} \rightarrow ((Ass^i_{er} \rightarrow Q_{l_{i,2}}) \wedge C^i_{er}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, h \mapsto v, \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Since $\#A > 0$, the first implication of Ass^i_{er} satisfies $isRrReq(last(A))$, $isUniquePID(last(A).spid)$, and $isValidInd(last(A).cap)$. Furthermore, we have $\sigma' \models |icTree| > 2$ because $isValidInd(last(A).cap)$ indicates that there is an intermediate indicator in addition to the root and the parent capability inside the tree. Thus, this verification holds.
- $\models Q_{l_{i,2}} \ \wedge \ Ass^i_{er} \rightarrow Q_{l_{i,3}} \circ f_{i,1}$. In this internal transition, function $f_{i,1}$ copies the intermediate indicator inside the tree, which $last(A).cap$ points to it, removes it from the tree, and assigns the copied intermediate indicator to variable $iIndCap$ when it returns. Thus, this verification holds.
- $\models Q_{l_{i,3}} \ \wedge \ Ass^i_{er} \rightarrow Q_{l_{i,4}} \circ f_{i,2}$. In this internal transition, function $f_{i,2}$ creates a capability-revocation request for program R_{er} and assigns it to variable y_i, such that $isRrReq(y_i) = true$ and $y_i.spid = last(A).spid$. Furthermore, we have $isUniquePID(y_i.spid) = true$ because $isUniquePID(last(A).spid) = true$ and $y_i.spid = last(A).rpid$. In addition, this transition satisfies $isValidInd(y_i.cap)$ because we have $isValidInd(indCap)$ and $y_i.cap = indCap$. Thus, this verification holds.
- $\models Q_{l_{i,4}} \wedge Ass^i_{er} \rightarrow ((Ass^i_{er} \rightarrow Q_{l_{i,5}}) \wedge C^i_{er}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y_i)))$. In this output transition, program I_{er} sends y_i through channel C. C^i_{er} holds because $last(C) = \sigma(y_i)$ and $isRrReq(y_i)$, $isUniquePID(y_i.spid)$, and $isValidInd(y_i.cap)$. Hence, this verification holds.
- $\models Q_{l_{i,5}} \wedge Ass^i_{er} \rightarrow ((Ass^i_{er} \rightarrow Q_{l_{i,6}}) \wedge C^i_{er}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z_i, h \mapsto v, \sigma(h).(D, v))$ for arbitrary value v. In this input transition, a communication through channel D took place, and function g assigns the received value to z_i. Since $\#D > 0$, Ass^i_{er} implies that $\sigma' \models isRrConfRes(z_i)$. Thus, this verification holds.
- $\models Q_{l_{i,6}} \ \wedge \ Ass^i_{er} \rightarrow Q_{l_{i,7}} \circ f_{i,3}$. In this internal transition, function $f_{i,3}$ creates a external revocation-confirmation response for the process and assigns it to w_i such that $isRrConfRes(w_i)$. Hence, this verification holds.
- $\models Q_{l_{i,7}} \wedge Ass^i_{er} \rightarrow ((Ass^i_{er} \rightarrow Q_{l_{i,1}}) \wedge C^i_{er}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(w_i)))$. In this output transition, program I_{er} sends w_i through channel B. Commitment C^i_{er} holds because $\sigma \models isRrConfRes(w_i)$. Hence, this verification holds.

Since $Q_{t^i_{er}} \rightarrow \psi^i_{er}$, the post-condition of program I_{er} is satisfied.

5.7.2 External Revocation - Resource Controller

Program R_{er} receives a capability-revocation request from program I_{er} via channel B. In the next step, program R_{er} removes the delegated capability and all re-delegated capabilities in its sub-tree. Finally, it sends a revocation-confirmation response toward program I_{er}. The functions of R_{er} are defined as follow:

$$
\begin{aligned}
init^r_{er}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))),\\
f_{r,1}(\sigma) &= (\sigma : subtreeList \mapsto revoke(\sigma(rootPtr), \sigma(x_r.cap))),\\
f_{r,2}(\sigma) &= (\sigma : isListEmpty \mapsto checkList(\sigma(subtreeList))),\\
f_{r,3}(\sigma) &= (\sigma : nextCap \mapsto getNextCap(\sigma(subtreeList))),\\
f_{r,4}(\sigma) &= (\sigma : revokeCap(\sigma(nextCap))),\\
f_{r,5}(\sigma) &= (\sigma : y_r \mapsto genRevConfRes(\sigma(x_r)))
\end{aligned}
$$

The A-C formula for program R_{er} is as follows:

$$\vdash \langle Ass^r_{er},\ C^r_{er} \rangle\ \{\varphi^r_{er}\}\ R_{er}\ \{\psi^r_{er}\}$$

where the formal definition of φ^r_{er}, ψ^r_{er}, Ass^r_{er}, and C^r_{er} are as follows:

$$
\begin{aligned}
\varphi^r_{er} &\stackrel{\text{def}}{=} \#C = \#D = 0\ \wedge\ |rcTree| \geqslant 1\\
\psi^r_{er} &\stackrel{\text{def}}{=} false\\
Ass^r_{er} &\stackrel{\text{def}}{=} \#C > 0 \rightarrow\\
&\quad (isRrReq(last(B))\ \wedge\ isUniquePID(last(B).spid)\ \wedge\\
&\quad isValidInd(last(B).cap))\\
C^r_{er} &\stackrel{\text{def}}{=} \#C = \#D > 0 \rightarrow isRrConfRes(last(D))
\end{aligned}
$$

The assertion network for R_{er} is as follows:

$$
\begin{aligned}
Q_{s^r_{er}} &\stackrel{\text{def}}{=} \#C = \#D = 0\ \wedge\ |rcTree \geqslant 2\\
Q_{l_{r,1}} &\stackrel{\text{def}}{=} \#C = \#D\ \wedge\ |rcTree| \geqslant 2\\
Q_{l_{r,2}} &\stackrel{\text{def}}{=} \#D = (\#C - 1)\ \wedge\ last(C) = x_r\ \wedge\ |rcTree| > 2\ \wedge\\
&\quad isRrReq(last(C))\ \wedge\ isUniquePID(last(C).spid)\ \wedge\\
&\quad isValidInd(last(C).cap)\\
Q_{l_{r,3}} &\stackrel{\text{def}}{=} \#D = (\#C - 1)\ \wedge\ last(C) = x_r\ \wedge\ |rcTree| \geqslant 2\\
Q_{l_{r,4}} &\stackrel{\text{def}}{=} \#D = (\#C - 1)\ \wedge\ last(C) = x_r\ \wedge\ |rcTree| \geqslant 2\ \wedge\ \neg isListEmpty\\
Q_{l_{r,5}} &\stackrel{\text{def}}{=} \#D = (\#C - 1)\ \wedge\ last(C) = x_r\ \wedge\ |rcTree| \geqslant 2\ \wedge\ isListEmpty\ \wedge
\end{aligned}
$$

$$isRrConfRes(y_r)$$

$$Q_{t^r_{er}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^r_{er}, commitment C^r_{er}, and channels C and D.

- $\models Q_{s^r_{er}} \rightarrow C^r_{er}$ follows from the above definitions.
- $\models Q_{s^r_{er}} \wedge Ass^r_{er} \rightarrow Q_{l_{r,1}} \circ init^r_{er}$. In this internal transition, the size of the intermediate-capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{r,1}} \wedge Ass^r_{er} \rightarrow ((Ass^r_{er} \rightarrow Q_{l_{r,2}}) \wedge C^r_{er}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_r, h \mapsto v, \sigma(h).(C, v))$ for arbitrary value v. In this input transition, just communication through channel C took place, and function g assigns the received value to x_r. Since $\#C > 0$, the first implication of Ass^r_{er} satisfies $isRrReq(last(C))$, $isUniquePID(last(C).spid)$, and $isValidInd(last(C).cap)$. Furthermore, we have $\sigma' \models |rcTree| > 2$ because $isValidInd(last(C).cap)$ indicates that there is a delegated resource capability in addition to the root and the parent capability inside the tree. Thus, this verification holds.
- $\models Q_{l_{r,2}} \wedge \neg isListEmpty \wedge Ass^r_{er} \rightarrow Q_{l_{r,3}} \circ (f_{r,2} \circ f_{r,1})$. In this internal transition, function $f_{i,1}$ revokes the resource capability inside the tree, which $last(C).cap$ points to it. In addition, it returns the sub-tree of the resource capability as a list. Thus, we have $\sigma' \models |rcTree| \geqslant 2$. In the next step, function $f_{r,2}$ checks if the list is empty. Hence, this verification holds.
- $\models Q_{l_{r,3}} \wedge \neg isListEmpty \wedge Ass^r_{er} \rightarrow Q_{l_{r,4}} \circ (f_{r,4} \circ f_{r,3})$. In this internal transition, the $\neg isListEmpty$ condition implies that $\sigma' \models \neg isListEmpty$. Hence, function $f_{i,3}$ gets the next resource capability that program R_{er} should revokes . It assigns the next resource capability to variable $nextCap$. Thus, this verification holds.
- $\models Q_{l_{r,4}} \wedge Ass^r_{er} \rightarrow Q_{l_{r,3}} \circ f_{r,2}$. In this internal transition, function $f_{r,2}$ checks if the list is empty. Hence, this verification holds.
- $\models Q_{l_{r,3}} \wedge isListEmpty \wedge Ass^r_{er} \rightarrow Q_{l_{r,5}} \circ f_{r,5}$. In this internal transition, the $isListEmpty$ condition implies that $\sigma' \models isListEmpty$. Hence, function $f_{r,5}$ generates a revocation-confirmation response and assigns it to variable y_r. Hence, this verification holds.
- $\models Q_{l_{r,5}} \wedge Ass^r_{er} \rightarrow ((Ass^r_{er} \rightarrow Q_{l_{r,1}}) \wedge C^r_{er}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(D, \sigma(y_r)))$. In this output transition, program R_{er} sends y_r through channel D. Commitment C^r_{er} holds because $\sigma \models isRrConfRes(y_r)$. Hence, this verification holds.

  Since $Q_{t^r_{er}} \rightarrow \psi^r_{er}$, the post-condition of program R_{er} is satisfied.

5.7.3 External Revocation - Parallel Composition

This section presents the parallel composition of the external-revocation components based on the described rule in 1. Consider the parallel composition $R_{er} \parallel I_{er}$. Program R_{er} satisfies the assumption-commitment pair $(Ass^r_{er},\ C^r_{er})$ as follows:

$$\vdash \langle Ass^r_{er},\ C^r_{er}\rangle\ \{\varphi^r_{er}\}\ R_{er}\ \{\psi^r_{er}\}$$

where the formal definition of φ^r_{er}, ψ^r_{er}, Ass^r_{er}, and C^r_{er} are as follows:

$$\begin{array}{lll}
\varphi^r_{er} & \stackrel{\text{def}}{=} & \#C = \#D = 0\ \wedge\ |rcTree| \geqslant 1 \\
\psi^r_{er} & \stackrel{\text{def}}{=} & false \\
Ass^r_{er} & \stackrel{\text{def}}{=} & \#C > 0 \rightarrow \\
 & & (isRrReq(last(B))\ \wedge\ isUniquePID(last(B).spid) \wedge \\
 & & isValidInd(last(B).cap)) \\
C^r_{er} & \stackrel{\text{def}}{=} & \#C = \#D > 0 \rightarrow isRrConfRes(last(D))
\end{array}$$

Program I_{er} satisfies the assumption-commitment pair $(Ass^i_{er},\ C^i_{er})$ as follows:

$$\vdash \langle Ass^i_{er},\ C^i_{er}\rangle\ \{\varphi^i_{er}\}\ I_{er}\ \{\psi^i_{er}\}$$

where the formal definition of φ^i_{er}, ψ^i_{er}, Ass^i_{er}, and C^i_{er} are as follows:

$$\begin{array}{lll}
\varphi^i_{er} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0\ \wedge\ |icTree| \geqslant 1 \\
\psi^i_{er} & \stackrel{\text{def}}{=} & false \\
Ass^i_{er} & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow \\
 & & isRrReq(last(A))\ \wedge\ isUniquePID(last(A).spid))) \\
 & & isValidInd(last(A).cap)) \wedge \\
 & & (\#C = \#D > 0 \rightarrow isRrConfRes(last(D))) \\
C^i_{er} & \stackrel{\text{def}}{=} & (\#A = \#B = \#C = \#D > 0 \rightarrow isRrConfRes(last(B))) \wedge \\
 & & (\#C > 0 \rightarrow isRrConfRes(last(C)))
\end{array}$$

By applying the parallel composition rule, we deduce as follows:

$$\begin{array}{c}
\vdash \langle Ass_{er},\ C_{er}\rangle \\
\{\varphi_{er}\}\ R_{er} \parallel I_{er}\ \{\psi_{er}\}
\end{array}$$

where the formal definition of φ_{er}, ψ_{er}, Ass_{er}, and C_{er} are as follows:

$$\begin{array}{lll}
\varphi_{er} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0\ \wedge\ |rcTree| \geqslant 1 \wedge \\
 & & |icTree| \geqslant 1
\end{array}$$

$$
\begin{aligned}
\psi_{er} &\stackrel{\text{def}}{=} false \\
Ass_{er} &\stackrel{\text{def}}{=} \#A > 0 \rightarrow \\
&\quad (isRrReq(last(A)) \ \wedge \ isUniquePID(last(A).spid) \wedge \\
&\quad isValidInd(last(A).cap)) \\
C_{er} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D > 0 \rightarrow isRrConfRes(last(B))
\end{aligned}
$$

□

5.8 Fifth Lemma: Internal Revocation

The fifth lemma indicates that the internal revocation of a valid p-cap by the delegator process will invalidate the delegated p-cap to another process inside the same node. Furthermore, the revocation operation encompasses all the re-delegated capabilities from the first delegated capability. Consequently, the resource controller participates in the internal-revocation operation if one of the re-delegations includes a external delegation. We formally define the lemma and proof it.

Lemma 5. *Given a process with a unique PID in a process node, an intermediate controller in the same node, and a resource controller inside a resource node, revoking a delegated p-cap to another process inside the same node invalidates the delegated p-cap and all its internal and external re-delegations.*

Proof. We use the assumption-commitment technique to prove the lemma.

When a process receives a internally delegated capability, it may perform a internal or external re-delegation. This re-delegation can include both internal and external delegations. Hence, the intermediate and resource controllers must participate in revoking the delegated capability and all the re-delegated capability in its sub-tree. To perform the revocation, program Ir collects the delegated capabilities in a list. Then, it iterates through the list and revokes the capabilities.

We model the intermediate controller at the delegating-process side and the resource controllers in a external-revocation operation by programs $I_{rr} \stackrel{\text{def}}{=} (L^i_{ir}, T^i_{ir}, s^i_{ir}, t^i_{ir})$ and $R_{rr} \stackrel{\text{def}}{=} (L^r_{rr}, T^r_{rr}, s^r_{rr}, t^r_{rr})$, respectively. The programs and the process exchange messages via synchronous channels to handle internal-revocation and possible external-revocation requests. Figure 13 depicts these two programs and their syntactic interfaces. Program R_{rr} operates as explained in 5.7.1. Figures 16 and 14b illustrate sequential diagrams of programs I_{ir} and R_{rr}, respectively. In the next step, we prove that when the process sends a internal-revocation request, programs I_{ir} and R_{rr} revoke the delegated capability along with all the re-delegated capabilities in its sub-tree.

Program I_{ir} receives a internal capability-revocation request from program P_{ir} via channel A and removes the delegated capability and all its internal or external re-delegated capabilities from the capability tree with the participation of the resource controller. Finally, it returns revocation confirmation response to the process via channel B. The conditions and functions of I_{ir} are as follows:

$$init^i_{ir}(\sigma) = \ (\sigma : rootPtr, isListEmpty \mapsto$$

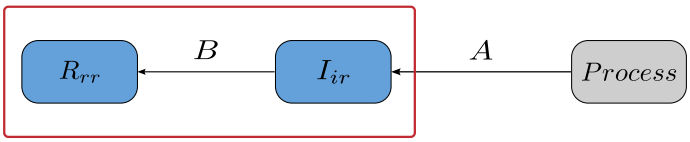


Figure 15: Syntactic Interfaces of the Internal-revocation Programs.

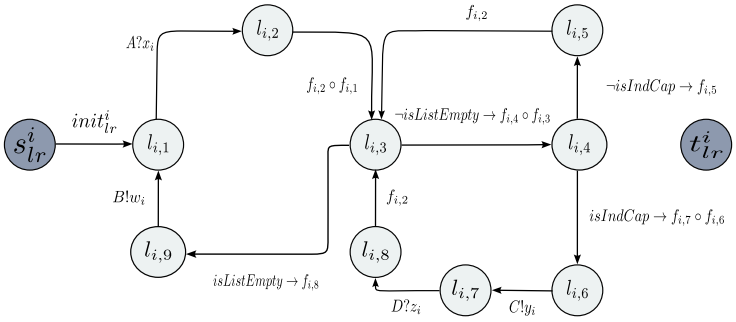


Figure 16: Internal Revocation - Program I_{ir}.

$$
\begin{aligned}
& \quad getRoot(\sigma(icTree), true)), \\
f_{i,1}(\sigma) &= (\sigma : capsList \mapsto \\
& \quad getDelegatedCapsList(\sigma(rootPtr), \sigma(x_i.cap))), \\
f_{i,2}(\sigma) &= (\sigma : isListEmpty \mapsto IsCapsListEmpty(\sigma(capsList))), \\
f_{i,3}(\sigma) &= (\sigma : nextCap \mapsto getNextCap(\sigma(capsList)), \\
f_{i,4}(\sigma) &= (\sigma : isIndCap \mapsto isIndCapability(\sigma(nextCap))), \\
f_{i,5}(\sigma) &= (\sigma : revokeCap(\sigma(nextCap))), \\
f_{i,6}(\sigma) &= (\sigma : iIndCap \mapsto revokeInd(\sigma(nextCap))), \\
f_{i,7}(\sigma) &= (\sigma : y_i \mapsto genRrReq(\sigma(x_i), \sigma(iIndCap))), \\
f_{i,8}(\sigma) &= (\sigma : w_i \mapsto genRevConfRes(\sigma(z_i))))
\end{aligned}
$$

The A-C formula for program I_{ir} is as follows:

$$\vdash \langle Ass^i_{ir},\ C^i_{ir} \rangle\ \{\varphi^i_{ir}\}\ I_{ir}\ \{\psi^i_{ir}\}$$

The formal definition of φ^i_{ir}, ψ^i_{ir}, As^i_{ir}, and C^i_{ir} are as follows:

$$
\begin{aligned}
\varphi^i_{ir} &\stackrel{\text{def}}{=} & & \#A = \#B = \#C = \#D = 0 \ \wedge\ |icTree| \geqslant 1 \\
\psi^i_{ir} &\stackrel{\text{def}}{=} & & false \\
Ass^i_{ir} &\stackrel{\text{def}}{=} & & (\#A > 0 \rightarrow \\
& & & \quad isLrReq(last(A)) \ \wedge\ isValidInd(last(A).cap))) \wedge \\
& & & (\#C = \#D > 0 \rightarrow isRrConfRes(last(D)))
\end{aligned}
$$

$$C^i_{ir} \quad \stackrel{\text{def}}{=} \quad \#A = \#B > 0 \rightarrow isLrConfRes(last(B))$$

The assertion network for I_{ir} is as follows:

$$Q_{s^i_{ir}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree| \geqslant 1$$

$$Q_{l_{i,1}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D \ \wedge \ isListEmpty$$

$$Q_{l_{i,2}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ isLrReq(last(A)) \wedge$$
$$isUniquePID(last(A).spid) \ \wedge \ isValidInd(last(A).cap) \ \wedge \ isListEmpty$$

$$Q_{l_{i,3}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i$$

$$Q_{l_{i,4}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ \neg isListEmpty$$

$$Q_{l_{i,5}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ \neg isIndCap$$

$$Q_{l_{i,6}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \wedge$$
$$isIndCap \ \wedge \ isValidInd(iIndCap)$$
$$isRrReq(y_i) \ \wedge \ isUniquePID(y_i.spid) \ \wedge \ isValidInd(y_i.cap)$$

$$Q_{l_{i,7}} \stackrel{\text{def}}{=} \#B = \#D = (\#A - 1) = (\#C - 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i$$

$$Q_{l_{i,8}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \ \wedge \ l$$
$$ast(D) = z_i \ \wedge \ isRrConfRes(z_i)$$

$$Q_{l_{i,9}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ isListEmpty \ \wedge \ isLrConfRes(z_i)$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^i_{ir}, commitment C^i_{ir}, and channels A, B, C, and D.

- $\models Q_{s^i_{ir}} \rightarrow C^i_{ir}$ follows from the above definitions.
- $\models Q_{s^i_{ir}} \ \wedge \ Ass^i_{ir} \rightarrow Q_{l_{i,1}} \circ init^i_{ir}$. In this internal transition, function $init^i_{ir}$ just assigns *true* to variable *isListEmpty*. Hence, this verification holds.
- $\models Q_{l_{i,1}} \wedge Ass^i_{ir} \rightarrow ((Ass^i_{ir} \rightarrow Q_{l_{i,2}}) \wedge C^i_{ir}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, h \mapsto v, \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Since $\#A > 0$, the first implication of Ass^i_{ir} satisfies $isLrReq(last(A))$, $isUniquePID(last(A).spid)$, and $isValidInd(last(A).cap)$. Thus, this verification holds.
- $\models Q_{l_{i,2}} \ \wedge \ Ass^i_{ir} \rightarrow Q_{l_{i,3}} \circ (f_{i,2} \circ f_{i,1})$. In this internal transition, function $f_{i,1}$ initialize the variable *capsList* with the delegated capability, which $last(A).cap$ points to it, and all the re-delegated capabilities in its sub-tree. Afterward, function $f_{i,2}$ checks if the list is empty. Thus, this verification holds.

- $\models Q_{l_{i,3}} \wedge \neg isListEmpty \wedge Ass^i_{ir} \rightarrow Q_{l_{i,4}} \circ (f_{i,4} \circ f_{i,3})$. In this internal transition, the condition implies that there is another capability in the list that the intermediate controller should revoke it. Function $f_{i,3}$ extracts this capability from the list, and function $f_{i,4}$ checks if it is a externally delegated capability. Hence, this verification holds.

- $\models Q_{l_{i,4}} \wedge \neg isIndCap \wedge Ass^i_{ir} \rightarrow Q_{l_{i,5}} \circ f_{i,5}$. The condition implies that the intermediate controller can revoke it internally in this internal transition. Hence, function $f_{i,5}$ revokes the capability internally by invalidating and removing it from the capability tree. Thus, this verification holds.

- $\models Q_{l_{i,5}} \wedge Ass^i_{ir} \rightarrow Q_{l_{i,3}} \circ f_{i,2}$. In this internal transition, function $f_{i,2}$ checks if the list is empty. Hence, this verification holds.

- $\models Q_{l_{i,4}} \wedge isIndCap \wedge Ass^i_{ir} \rightarrow Q_{l_{i,6}} \circ (f_{i,7} \circ f_{i,6})$. The condition indicates that the intermediate and resource controller should collaborate because it is a externally delegated capability. Hence, function $f_{i,6}$ copies the intermediate indicator capability, removes it from the tree, and assigns the copied intermediate indicator to variable $iIndCap$ when it returns. Afterward, function $f_{i,7}$ creates a capability-revocation request for program R_{rr} and assigns it to variable y_i, such that $isRevReq(y_i) = true$ and $y_i.spid = last(A).spid$. In addition, $\sigma' \models isUniquePID(y_i.spid)$ because it is equal to the receiving process id from the process, and Ass^i_{ir} implies that $isUniquePID(last(A).spid)$ Thus, this verification holds.

- $\models Q_{l_{i,6}} \wedge Ass^i_{ir} \rightarrow ((Ass^i_{ir} \rightarrow Q_{l_{i,7}}) \wedge C^i_{ir}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y_i)))$. In this output transition, program I_{ir} sends y_i through channel C. C^i_{ir} holds because $last(C) = \sigma(y_i)$, and from $Q_{l_{i,7}}$, we have $isRevReq(y_i)$, $isUniquePID(y_i.spid)$, and $isValidInd(y_i.cap)$. Hence, this verification holds.

- $\models Q_{l_{i,7}} \wedge Ass^i_{ir} \rightarrow ((Ass^i_{ir} \rightarrow Q_{l_{i,8}}) \wedge C^i_{ir}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z_i, h \mapsto v, \sigma(h).(D, v))$ for arbitrary value v. In this input transition, a communication through channel D took place, and function g assigns the received value to z_i. Since $\#D > 0$, Ass^i_{ir} implies that $\sigma' \models isRrConfRes(z_i)$. Thus, this verification holds.

- $\models Q_{l_{i,8}} \wedge Ass^i_{ir} \rightarrow Q_{l_{i,3}} \circ f_{i,2}$. In this internal transition, function $f_{i,2}$ checks if the list is empty. Hence, this verification holds.

- $\models Q_{l_{i,3}} \wedge isListEmpty \wedge Ass^i_{ir} \rightarrow Q_{l_{i,9}} \circ (f_{i,8} \circ unlockTree)$. In this internal transition, the condition implies that the intermediate controller has revoked all the delegated capabilities from the list. Afterward, function $f_{i,8}$ creates a revocation-confirmation response for the process and assigns it to w_i such that $isLrConfRes(w_i)$. Thus, this verification holds.

- $\models Q_{l_{i,9}} \wedge Ass^i_{ir} \rightarrow ((Ass^i_{ir} \rightarrow Q_{l_{i,1}}) \wedge C^i_{ir}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(w_i)))$. In this output transition, program I_{ir} sends w_i through channel B. Commitment C^i_{ir} holds because $\sigma \models isLrConfRes(w_i)$. Hence, this verification holds.

  Since $Q_{t^i_{ir}} \rightarrow \psi^i_{ir}$, the post-condition of program I_{ir} is satisfied.

5.8.1 Internal Revocation - Parallel Composition

This section presents the parallel composition of the external-revocation components based on the described rule in 1. Consider the parallel composition $R_{rr} \parallel I_{ir}$. Program R_{rr} satisfies the assumption-commitment pair $(Ass^r_{rr},\ C^r_{rr})$ as follows:

$$\vdash \langle Ass^r_{rr},\ C^r_{rr}\rangle\ \{\varphi^r_{rr}\}\ R_{rr}\ \{\psi^r_{rr}\}$$

where the formal definition of φ^r_{rr}, ψ^r_{rr}, Ass^r_{rr}, and C^r_{rr} are as follows:

$$\begin{aligned}
\varphi^r_{rr} &\stackrel{\text{def}}{=} \#C = \#D = 0\ \wedge\ |rcTree| \geqslant 1\\
\psi^r_{rr} &\stackrel{\text{def}}{=} false\\
Ass^r_{rr} &\stackrel{\text{def}}{=} \#C > 0 \rightarrow\\
&\quad (isRrReq(last(B))\ \wedge\ isUniquePID(last(B).spid) \wedge\\
&\quad\ isValidInd(last(B).cap))\\
C^r_{rr} &\stackrel{\text{def}}{=} \#C = \#D > 0 \rightarrow isRrConfRes(last(D))
\end{aligned}$$

Program I_{rr} satisfies the assumption-commitment pair $(Ass^i_{ir},\ C^i_{ir})$ as follows:

$$\vdash \langle Ass^i_{ir},\ C^i_{ir}\rangle\ \{\varphi^i_{ir}\}\ I_{ir}\ \{\psi^i_{ir}\}$$

The formal definition of φ^i_{ir}, ψ^i_{ir}, As^i_{ir}, and C^i_{ir} are as follows:

$$\begin{aligned}
\varphi^i_{ir} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0\ \wedge\ |icTree| \geqslant 1\\
\psi^i_{ir} &\stackrel{\text{def}}{=} false\\
Ass^i_{ir} &\stackrel{\text{def}}{=} (\#A > 0 \rightarrow\\
&\quad\ isLrReq(last(A))\ \wedge\ isValidInd(last(A).cap))) \wedge\\
&\quad (\#C = \#D > 0 \rightarrow isRrConfRes(last(D)))\\
C^i_{ir} &\stackrel{\text{def}}{=} \#A = \#B > 0 \rightarrow isLrConfRes(last(B))
\end{aligned}$$

By applying the parallel composition rule, we deduce as follows:

$$\begin{gathered}
\vdash \langle Ass_{ir},\ C_{ir}\rangle\\
\{\varphi_{ir}\}\ R_{ir} \parallel I_{ir}\ \{\psi_{ir}\}
\end{gathered}$$

where the formal definition of φ_{ir}, ψ_{ir}, Ass_{ir}, and C_{ir} are as follows:

$$\begin{aligned}
\varphi_{ir} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0\ \wedge\ |rcTree| \geqslant 1\ \wedge\\
&\quad\ |icTree| \geqslant 1\\
\psi_{ir} &\stackrel{\text{def}}{=} false
\end{aligned}$$

$$Ass_{ir} \stackrel{\text{def}}{=} \#A > 0 \rightarrow (isLrReq(last(A)) \ \wedge \ isValidInd(last(A).cap))$$

$$C_{ir} \stackrel{\text{def}}{=} (\#A = \#B > 0 \wedge \#C = \#D > 0) \rightarrow isLrConfRes(last(B))$$

□

5.8.2 Capability Safety

Capability safety implies that a process can only obtain its capabilities through a resource allocation or a capability delegation and only employ its valid capabilities.

Definition 22. *Capability System. A capability system is a tuple comprising the following:*

- *a capability set* $\mathbb{C} = \{\mathbb{C}_r \cup \mathbb{C}_i \cup \mathbb{C}_p\}$*;*
- *a function:* $rsrc: \mathbb{C} \rightarrow \mathbb{R}$
- *a function:* $priv: \mathbb{C} \rightarrow 2^{\mathbb{V}}$
- *a function:* $pCaps: \mathbb{P} \rightarrow 2^{\mathbb{C}_p}$
- *a function:* $cAuth: \mathbb{C} \rightarrow \mathbb{R} \times 2^{\mathbb{V}}$
- *a function:* $pAuth: \mathbb{P} \rightarrow 2^{\mathbb{R}} \times 2^{\mathbb{V}}$

The tuple must satisfy the following conditions to denote a valid capability system: $\forall \ c_p \in \mathbb{C}_p, c_i \in \mathbb{C}_i, c_r \in \mathbb{C}_r, T_i \in \mathbb{T}_i, \ and \ T_r \in \mathbb{T}_r$

(a) $c_p \mapsto c_i \Rightarrow cAuth(c_p) \subseteq cAuth(c_i)$

(b) $c_i \mapsto c_r \Rightarrow cAuth(c_i) \subseteq cAuth(c_r)$

(c) $isInTree(T_i, c_i, c_i') \ \Rightarrow \ cAuth(c_i') \subseteq cAuth(c) \ \ s.t. \ \ c_i \neq c_i^{root}$

(d) $isInTree(T_r, c_r, c_r') \Rightarrow \ cAuth(c_r') \subseteq cAuth(c_r)$

The valid-capability set of a process indicates the legitimately acquired capabilities by the process that are still valid; thus, the process can employ them for read/write operations. We formally define this set as follows:

$$validCaps(p) \subseteq pCaps(p): (\forall c_p \in pCaps(p) \ \ s.t. \ isValidProCap(c_p) \Rightarrow \ c_p \in validCaps(p))$$

Now, we define the capability-safety property and prove that our capability-based access-control system is capability safe.

Definition 23. *Capability Safety. An access-control system is capability safe, concerning the capability system* $(\mathbb{C}, rsrc, priv, pCap, cAuth, pAuth)$*, if the following condition holds for all processes* $p \in \mathbb{P}$:

- $$rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$$

Where $rsrcSet$ indicates accessible resources by the process.

Proof. The proof proceeds by induction on the structure of the process's capability set. The proof consists of one base case and two inductive cases.

- **Base Case:** The capability set is empty.
  In this case, the process possesses no valid capability; thus, the process can access no resources.

- **Inductive Case One:** $validCaps(p)$ contains the current process's valid capabilities; hence, we have:

  $$rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$$

  Based on Lemmas 1 to 3, the process receives a valid process capability, c_p^{new}, due to a resource-allocation or internal/external-delegation request. Its valid-capability set changes as follows:

  $$validCaps'(p) = validCaps(p) \cup c_p^{new}$$

  Hence, its new resource list transforms as follows:

  $$rsrcSet'(p) = rsrcSet(p) \cup rsrc(c_p^{new})$$

- **Inductive Case Two:** $validCaps(p)$ contains the current process's valid capabilities; hence, we have:

  $$rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$$

  Based on Lemmas 4 to 5, one of the process's capabilities, c_p^{old}, becomes invalid due to an internal/external revocation. Its valid-capability set changes as follows:

  $$validCaps'(p) = validCaps(p) \setminus c_p^{old}$$

  Thus, its new resource list transforms as follows:

  $$rsrcSet'(p) = rsrcSet(p) \setminus rsrc(c_p^{old})$$

□

5.8.3 Authority Safety

We claim that a distributed capability-based access-control system is authority safe if it addresses the following properties regarding the object-capability model:

- **An authority's owner obtained it through a capability delegation**: A process can acquire authority only if the owner of the authority delegates it to the process.
- **No authority amplification**: The capability owners can only delegate what authority they have.

Our access-control system guarantees both properties. Processes can only obtain capabilities through allocating resources or delegating capabilities by other processes. Furthermore, using well-formed capability trees, the access-control system ensures that no process can delegate an authority it does not own. We define the authority-safety property as follows:

Definition 24. *Authority Safety. A capability-based access-control system is authority safe, concerning the capability system* ($\mathbb{C}$, $rsrc$, $priv$, $pCap$, $cAuth$, $pAuth$)*, if the following condition holds for all processes* $p \in \mathbb{P}$:[1]

- $$pAuth(p) = \bigcup_{c_p \in validCaps(p)} cAuth(c_p)$$

Proof. The proof proceeds by induction on the structure of the process's capability set. The proof consists of one base case and two inductive cases.

- **Base Case:** The capability set is empty.
  In this case, the process possesses no valid capability; thus, the process can access no resources.
- **Inductive Case One:** $validCaps(p)$ contains the current process's valid capabilities; hence, we have:

  $$rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$$

  Based on Lemmas 1 to 3, the process receives a valid process capability, c_p^{new}, due to a resource-allocation or internal/external-delegation request. Its valid-capability set changes as follows:

  $$validCaps'(p) = validCaps(p) \cup c_p^{new}$$

  Hence, its authority transforms as follows:

  $$pAuth'(p) = pAuth(p) \cup cAuth(c_p^{new})$$

  In addition, because all the capability trees are well-formed, the authority of the delegated capability is always a subset of the orginal capability.

- **Inductive Case Two:** $validCaps(p)$ contains the current process's valid capabilities; hence, we have:

$$rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$$

Based on Lemmas 4 to 5, one of the process's capabilities, c_p^{old}, becomes invalid due to an internal/external revocation. Its valid-capability set changes as follows:

$$validCaps'(p) = validCaps(p) \setminus c_p^{old}$$

Thus, its authority transforms as follows:

$$pAuth'(p) = pAuth(p) \setminus pAuth(c_p^{old})$$

□

5.9 Isolation Property

The isolation property expresses that two processes cannot access to the same resource. This property guarantees that untrusted processes cannot access trusted processes' resources. We formally define this property as follows:

Definition 25. *Isolation Property. Given a set of Resources $R \subseteq \mathbb{R}$ and a set of processes $p_1, \cdots, p_k \in P \subseteq \mathbb{P}$, we have $Isolation(R, p_1, \cdots, p_k \in P)$ if:*

- $\forall\, i, j\colon (1 \leqslant i < j \leqslant k \;\wedge\; rsrcSet(p_i) \subseteq R \;\wedge\; rsrcSet(p_i) \subseteq R)$
  $\Rightarrow\; rsrcSet(p_i) \cap rsrcSet(p_j) = \emptyset$

Processes can isolate their resources by delegating no capability. However, this approach is error-prone because it relies on developers to ensure processes never delegate their capabilities. Therefore, we support the isolation automatically using new permission. Let $\mathbb{Y} = \{d\}$ be the capability-related permission set, where d denotes the delegation permission. When controllers generate capabilities due to allocating resources, they unset this permission. Thus, because processes cannot delegate their capabilities and our capability-based access-control system is capability and authority safe, it supports the isolation property.

Proof. The proof proceeds by induction on the structure of capability trees.

we start with the resource capability. The proof consists of one base case and three inductive cases.

- **Base Case:** The capability set is empty.
  In this case, the resource capability tree only contain the root capability; thus, it satisfies the isolation property.

- **Inductive Case One:** The capability contains more than one capability and it satisfies the isolation property. The resource controller receives a resource-allocation request. Based on 1–**??** the resource controller allocates the resource

which is free and adds its corresponding resource capability to tree. Furthermore, it unset the delegation permission, d, in the resource capability. Because the new capability does not have any overlap with the existing ones, the tree satisfies the isolation property. Moreover, it creates the corresponding intermediate capability, in which the delegation permission is unset.

- **Inductive Case Two:**The capability contains more than one capability and it satisfies the isolation property. The resource controller receives a external-delegation request. Because the delegation permission is unset in all the resource capabilities inside the tree, the resource controller rejects the request and does not change the tree. Thus, the tree satisfies the isolation property.

- **Inductive Case Three:** The capability contains more than one capability and it satisfies the isolation property. The resource controller receives a external-revocation request. However, because the delegation permission is unset in all the capabilities inside the tree, none of them have a delegation-hierarchy as its sub-tree. Thus, the resource controller reject the request because the intermediate capability inside the request points to an non-existing resource capability. Therefore, the tree does not change and it satisfies the isolation property.

Now, we proof the isolation property for the intermediate tree. The proof consists of one base case and three inductive cases.

- **Base Case:** The capability set is empty.
  In this case, the intermediate capability tree only contain the root capability; thus, it satisfies the isolation property.

- **Inductive Case One:** The capability contains more than one capability and it satisfies the isolation property. The intermediate controller receives a resource-allocation response and inserts it as the direct child of the tree's root. Because the resource in the received intermediate capability does not have overlap with any resources in the resource node, it does not have overlap with any capability in the resource node. Thus, it does not have any overlap with the existing intermediate capability inside the intermediate tree. Thus, the tree satisfies the isolation property.

- **Inductive Case Two:**The capability contains more than one capability and it satisfies the isolation property. The intermediate controller receives a internal-delegation request. Because the delegation permission is unset in all the intermediate capabilities inside the tree, the intermediate controller rejects the request and does not change the tree. Thus, the tree satisfies the isolation property.

- **Inductive Case Three:** The capability contains more than one capability and it satisfies the isolation property. The resource controller receives a internal-delegation request. However, because the delegation permission is unset in all the capabilities inside the intermediate tree, none of them have a delegation-hierarchy as its sub-tree. Thus, the resource controller reject the request because the process capability inside the request points to an non-existing intermediate capability. Hence, the tree does not change and it satisfies the isolation property.

Therefore, we proved that our access-control system supports the isolation property.
□

6 Use Case

MDC [32] provides direct access to the shared memory pool. The direct access to the shared memory introduces a vulnerability to the MDC's architecture. For example, imagine a process that uses the shared memory pool as the communication medium due to its direct and memory-speed persistence access; the process writes data on the shared memory and passes the data's reference to another process to communicate with it. However, a third process can read the communicated data by knowing the reference and violate the security or privacy of data. Therefore, MDC requires an access-control system to preserve the security and privacy of data in the shared memory pool.

We instantiates our capability-based access-control system for MDC as the use case and demonstrate how it handles requests and capabilities[1]. In our general-purpose system, programs directly connect to other programs via synchronous channels (as depicted in Figure 7). However, in MDC, they are part of the instantiated controllers. An instantiated controller integrates all the related components inside the controller and connects them to the rest of the system via *handler*s and bridges. In addition, we add a module to OS which connects processes to the intermediate controller. We assume adversaries cannot compromise this module. This section describes describes nodes and controllers in the instantiated system, their components, and their syntactic interfaces. Furthermore, we explain how they cooperate to control access.

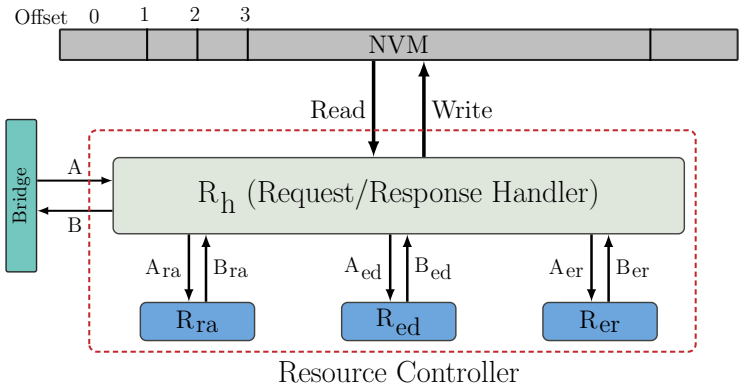
The new components change the environments of the existing components. In addition, new components commit to guaranteeing some of the previous assumptions. For example, the OS module checks the process ID in a request against the sender's PID because processes may act maliciously. Thus, it commits that each request contains a unique process ID, and it belongs to the sender. To prove the soundness of the instantiated system, we verify that it guarantees security properties. Thus, because the proofs of the properties depend on Lemmas 1 to 5, we must refine lemmas' proofs.

To refine Lemmas' proofs, we first verify the new modules regarding the assumption-commitment method. Afterward, we instantiate controllers by integrating the verified modules and applying the parallel composition rule (Rule 1). The soundness of the parallel composition rule depends on the validity of the A-C formulas of the integrated modules and the validity of the hypothesis of Rule 1 ($A \wedge C_1 \rightarrow A_2$, $A \wedge C_2 \rightarrow A_1$) for the joint channels between modules [16]. Finally, we verify the lemmas; thus, we prove that the instantiated system guarantees the security properties. Section *Use Case* of the Technical Report represents the whole proof.

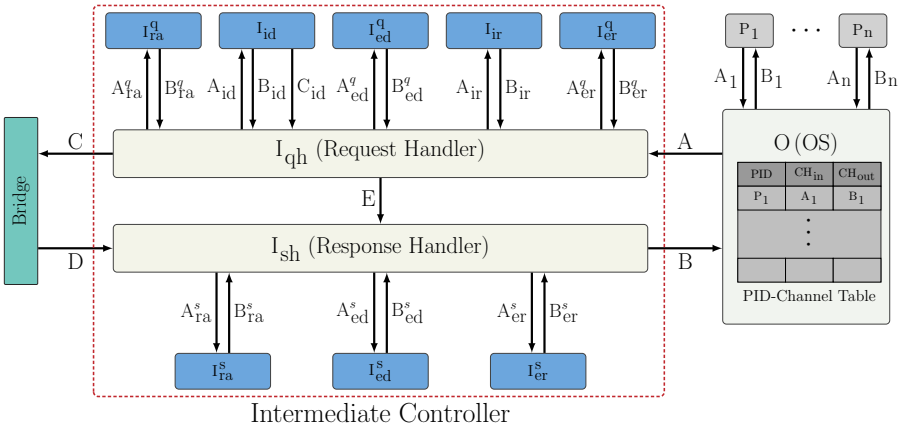
6.1 Resource Node

Figure 17a depicts a resource node in the instantiated system, including a resource controller and a byte-addressable NVM.

[1] **??** contains the capability structures of the instantiated system.



(a) Resource Node's Components and their Syntactic Interfaces.



(b) PE Node's Components and their Syntactic Interfaces.

Figure 17: Instantiated Access-Control System for MDC.

6.1.1 Resource Controller

The instantiated resource controller comprises four components: Request/Response handler (program R_h), Resource Allocation (R_{ra}), External Delegation (R_{ed}), and External Revocation (R_{er}). Each component has its assumptions and commitments. We define program R as the parallel execution of the programs in the controller as follows:

$$R \stackrel{\text{def}}{=} R_h \parallel R_{ra} \parallel R_{ed} \parallel R_{er}$$

With the A-C formula $\vdash \langle A^r, C^r \rangle : \{\varphi^r\} R \{\psi^r\}$. By applying the parallel composition rule, the commitments of the controller should be the commitments of all its components ($C_h \wedge C_{ra} \wedge C_{ed} \wedge C_{er}$). Furthermore, we should check the correctness of the controller's assumptions by checking the validity of the hypothesis ($A \wedge C_1 \rightarrow A_2$, $A \wedge C_2 \rightarrow A_1$) in Rule 1. It means, the controller's assumption and the handler's commitments should guarantee the assumption of programs R_{ra} (A_{ra}), R_{ed} (A_{ed}), and

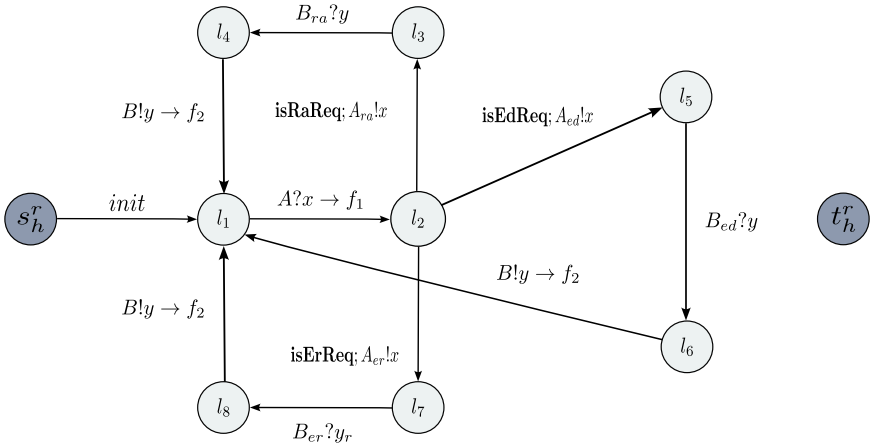


Figure 18: Request/Response-Handler Program of the Resource Controller.

R_{er} (A_{er}).

6.1.1.1 Request/Response Handler

We model the request/response Handler by program $R_h \stackrel{\text{def}}{=} (L_h^r, T_h^r, s_h^r, t_h^r)$. Program R_h receives requests from channel A and processes them in a *First In / First Out* (FIFO) order. It checks each request and commits to forwarding the request to the corresponding program. Program R_h assumes that each received request contain a unique process ID in the sender's PE node. Because each request contains a unique process ID and program R_h commits to forwarding it to the right program, which satisfies the programs' assumptions ($A^r \wedge C_h \rightarrow A_{ra,ed,er}$). Furthermore, these programs commit to returning valid responses (C_{ra}, C_{ed}, C_{er}). Thus, combining these commitments and assumption of channel A ($A^r \wedge C_{ra,ed,er}$) guarantees that R_h will receive valid responses, which it will send out through channel D. Therefore, the resource controller commits to return valid responses via channel D when it receives requests containing unique process IDs from channel A. Figure 18 depicts the state diagram of program R_h. The conditions and functions of program R_h are as follows:

$$
\begin{aligned}
init(\sigma) &= (\sigma : isRaReq,\ isEdReq,\ isErReq \mapsto false,\ false,\ false),\\
f_1(\sigma) &= (\sigma : isRaReq,\ isEdReq,\ isErReq \mapsto\\
&\qquad isRaReqCheck(\sigma(x)),\ isEdReqCheck(\sigma(x)),\ isErReqCheck(\sigma(x))),\\
f_2(\sigma) &= (\sigma : isRaReq,\ isEdReq,\ isErReq \mapsto false,\ false,\ false)
\end{aligned}
$$

The A-C formula for process R_{ra} is as follows:

$$\vdash \langle A_h^r\ C_h^r \rangle\ \{\varphi_h^r\}\ R_h\ \{\psi_h^r\}$$

where the formal definition of φ_r, ψ_r , A^r_{ra}, and C^r_{ra} are as follows:

$$\varphi^r_h \stackrel{\text{def}}{=} \#A = \#B = \#A_{ra} = \#B_{ra} = \#A_{ed} = \#B_{ed} = \#A_{er} = \#B_{er} = 0 \wedge \neg isRaReq \wedge \neg isEdReq \wedge \neg isErReq$$

$$\psi^r_h \stackrel{\text{def}}{=} false$$

$$A^r_h \stackrel{\text{def}}{=} (\#A > 0 \rightarrow isUniquePID(last(A).spid)) \\ ((\#A_{ra} = \#B_{ra} > 0) \rightarrow isValidCap(last(B_{ra}).cap)) \wedge \\ ((\#A_{ed} = \#B_{ed} > 0) \rightarrow isValidCap(last(B_{ed}).cap)) \wedge \\ ((\#A_{er} = \#B_{er} > 0) \rightarrow isValidCap(last(B_{er}).cap))$$

$$C^r_h \stackrel{\text{def}}{=} (\#A = \#B > 0) \rightarrow isValidCap(last(B).cap)) \wedge \\ ((\#A_{ra} > 0 \wedge \#A_{ra} \neq \#B_{ra}) \rightarrow isUniquePID(last(A_{ra}).spid)) \wedge \\ ((\#A_{ed} > 0 \wedge \#A_{ed} \neq \#B_{ed}) \rightarrow isUniquePID(last(A_{ed}).spid)) \wedge \\ ((\#A_{er} > 0 \wedge \#A_{er} \neq \#B_{er}) \rightarrow isUniquePID(last(A_{er}).spid))$$

The assertion network for R_h is as follows:

$$Q_{s_h} \stackrel{\text{def}}{=} \#A = \#B = 0 \wedge \#A_{ra} = \#B_{ra} = 0 \wedge \#A_{ed} = \#B_{ed} = 0 \wedge \#A_{er} = \#B_{er} = 0 \wedge \\ \neg isRaReq \wedge \neg isEdReq \wedge \neg isErReq$$

$$Q_1 \stackrel{\text{def}}{=} \#A = \#B \wedge \#A_{ra} = \#B_{ra} \wedge \#A_{ed} = \#B_{ed} \wedge \#A_{er} = \#B_{er} \wedge \\ \neg isRaReq \wedge \neg isEdReq \wedge \neg isErReq$$

$$Q_2 \stackrel{\text{def}}{=} (\#A - 1) = \#B \wedge \#A_{ra} = \#B_{ra} \wedge \#A_{ed} = \#B_{ed} \wedge \#A_{er} = \#B_{er} = 0 \wedge \\ \neg isRaReq \wedge \neg isEdReq \wedge \neg isErReq \wedge \\ last(A) = x \wedge isUniquePID(last(A).spid)$$

$$Q_3 \stackrel{\text{def}}{=} (\#A - 1) = \#B \wedge (\#A_{ra} - 1) = \#B_{ra} \wedge \#A_{ed} = \#B_{ed} \wedge \#A_{er} = \#B_{er} \wedge \\ isRaReq \wedge \neg isEdReq \wedge \neg isErReq \wedge last(A_{ra}) = x$$

$$Q_4 \stackrel{\text{def}}{=} (\#A - 1) = \#B \wedge \#A_{ra} = \#B_{ra} \wedge \#A_{ed} = \#B_{ed} \wedge \#A_{er} = \#B_{er} \wedge \\ isRaReq \wedge \neg isEdReq \wedge \neg isErReq \wedge \\ last(B_{ra}) = y \wedge isValidCap(last(B_{ra}).cap)$$

$$Q_5 \stackrel{\text{def}}{=} (\#A - 1) = \#B \wedge \#A_{ra} = \#B_{ra} \wedge (\#A_{ed} - 1) = \#B_{ed} \wedge \#A_{er} = \#B_{er} \wedge \\ \neg isRaReq \wedge isEdReq \wedge \neg isErReq \wedge last(A_{ed}) = x$$

$$Q_6 \stackrel{\text{def}}{=} (\#A - 1) = \#B \wedge \#A_{ra} = \#B_{ra} \wedge \#A_{ed} = \#B_{ed} \wedge \#A_{er} = \#B_{er} \wedge \\ \neg isRaReq \wedge isEdReq \wedge \neg isErReq \wedge \\ last(B_{ed}) = y \wedge isValidCap(last(B_{ed}).cap)$$

$$Q_7 \stackrel{\text{def}}{=} (\#A - 1) = \#B \wedge \#A_{ra} = \#B_{ra} \wedge \#A_{ed} = \#B_{ed} \wedge (\#A_{er} - 1) = \#B_{er} \wedge \\ \neg isRaReq \wedge \neg isEdReq \wedge isErReq \wedge last(A_{er}) = x$$

$$Q_8 \stackrel{\text{def}}{=} (\#A - 1) = \#B \ \wedge\ \#A_{ra} = \#B_{ra} \ \wedge\ \#A_{ed} = \#B_{ed} \ \wedge\ \#A_{er} = \#B_{er} \wedge \neg isRaReq \ \wedge\ \neg isEdReq \ \wedge\ isErReq \wedge last(B_{er}) = y \ \wedge\ isValidCap(last(B_{er}).cap)$$

$$Q_{t^r_{ra}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption A^r_h, commitment C^r_h, and channels A, B, A_{ra}, B_{ra}, A_{ed}, B_{ed}, A_{er}, and B_{er}.

- $\models Q_{s^r_h} \rightarrow C^r_h$ follows from the above definitions.
- $\models Q_{s^r_h} \ \wedge\ A^r_h \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does unset boolean variables *isRaReq*, *isEdReq*, and *isErReq*. Hence, this verification holds.
- $\models Q_{l_1} \wedge A^r_h \rightarrow ((A^r_h \rightarrow Q_{l_2}) \wedge C^r_h) \circ (f_1 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x,\ h \mapsto v,\ \sigma(h).(A, v))$ for arbitrary v. In this input transition, a communication through channel A just took place. Function g assigns the last received value from channel A to x. Because $\#A > 0$, the first implication of A^r_h satisfies $isUniquePID(last(A).spid)$ predicate. Function f_1 checks the variable x to find the request type and assigns boolean variables *isRaReq*, *isEdReq*, and *isErReq* accordingly. Thus, this verification holds.
- $\models Q_{l_2} \ \wedge\ isRaReq \ \wedge\ A^r_h \rightarrow ((A^r_h \rightarrow Q_{l_3}) \ \wedge\ C^r_h) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{ra}, \sigma(x)))$. In this output transition, program R_h sends x through channel A_{ra} toward program R_{ra}. From the transition's condition we have $isRaReq$. C^r_h holds because $\#A_{ra} > 0 \ \wedge\ \#A_{ra} \neq \#B_{ra}$ and $\sigma \models isUniquePID(x.spid)$. Hence, this verification holds.
- $\models Q_{l_3} \ \wedge\ A^r_h \rightarrow ((A^r_h \rightarrow Q_{l_4}) \ \wedge\ C^r_h) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y,\ h \mapsto v,\ \sigma(h).(B_{ra}, v))$ for arbitrary v. In this input transition, a communication through channel B_{ra} just took place and function g assigns the last received value to y. Because $\#A_{ra} = \#B_{ra} > 0$, the second implication of A^r_h satisfies $isValidCap(last(B_{ra}).cap)$. Thus, this verification holds.
- $\models Q_{l_4} \wedge A^r_h \rightarrow ((A^r_h \rightarrow Q_{l_1}) \wedge C^r_h) \circ (f_2 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y)))$. In this output transition, program R_h just sends out variable y through channel B. Furthermore, function f_2 does unset boolean variables *isRaReq*, *isEdReq*, and *isErReq*. C^r_h holds because $\#A = \#B > 0$, and $\sigma \models isValidCap(y.cap)$. Hence, this verification holds.
- $\models Q_{l_2} \ \wedge\ isEdReq \ \wedge\ A^r_h \rightarrow ((A^r_h \rightarrow Q_{l_5}) \ \wedge\ C^r_h) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{ed}, \sigma(x)))$. In this output transition, program R_h sends x through channel A_{ed} toward program R_{ed}. From the transition's condition we have $isEdReq$. C^r_h holds because $\#A_{ed} > 0 \ \wedge\ \#A_{ed} \neq \#B_{ed}$ and $\sigma \models isUniquePID(x.spid)$. Hence, this verification holds.
- $\models Q_{l_5} \ \wedge\ A^r_h \rightarrow ((A^r_h \rightarrow Q_{l_6}) \ \wedge\ C^r_h) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x,\ h \mapsto v,\ \sigma(h).(B_{ed}, v))$ for arbitrary v. In this input transition, a communication through channel B_{ed} just

took place and function g assigns the last received value to y. Because $\#A_{ed} = \#B_{ed} > 0$, the second implication of A_h^r satisfies $isValidCap(last(B_{ed}).cap)$. Thus, this verification holds.

- $\models Q_{l_6} \wedge A_h^r \rightarrow ((A_h^r \rightarrow Q_{l_1}) \wedge C_h^r) \circ (f_2 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y)))$. In this output transition, program R_h just sends out variable y through channel B. Furthermore, function f_2 does unset boolean variables $isRaReq$, $isEdReq$, and $isErReq$ C_h^r holds because $\#A = \#B > 0$, and $\sigma \models isValidCap(y.cap)$. Hence, this verification holds.

- $\models Q_{l_2} \wedge isEdReq \wedge A_h^r \rightarrow ((A_h^r \rightarrow Q_{l_7}) \wedge C_h^r) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{er}, \sigma(x)))$. In this output transition, program R_h sends x through channel A_{er} toward program R_{ed}. From the transition's condition we have $isErReq$. C_h^r holds because $\#A_{er} > 0 \wedge \#A_{er} \neq \#B_{ed}$ and $\sigma \models isUniquePID(x.spid)$. Hence, this verification holds.

- $\models Q_{l_7} \wedge A_h^r \rightarrow ((A_h^r \rightarrow Q_{l_8}) \wedge C_h^r) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x,\ h \mapsto v,\ \sigma(h).(B_{er}, v))$ for arbitrary v. In this input transition, a communication through channel B_{er} just took place and function g assigns the last received value to y. Because $\#A_{er} = \#B_{er} > 0$, the second implication of A_h^r satisfies $isValidCap(last(B_{er}).cap)$. Thus, this verification holds.

- $\models Q_{l_8} \wedge A_h^r \rightarrow ((A_h^r \rightarrow Q_{l_1}) \wedge C_h^r) \circ (f_2 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y)))$. In this output transition, program R_h just sends out variable y through channel B. Furthermore, function f_2 does unset boolean variables $isRaReq$, $isEdReq$, and $isErReq$ C_h^r holds because $\#A = \#B > 0$, and $\sigma \models isValidCap(y.cap)$. Hence, this verification holds.

Since $Q_{t_h^r} \rightarrow \psi_h^r$, the post-condition of program R_h is satisfied.

6.2 PE Node

Figure 17b illustrates a PE node in the instantiated system, comprising an intermediate controller, an operating system (OS), and running processes in the node.

6.2.1 Operating System

OS acts as a mediator between processes and the intermediate controller. We model the operating system in the PE node by program $O \stackrel{\text{def}}{=} (L^o, T^o, s^o, t^o)$. It has two tasks: assigning unique ID/communication channels to each process and forwarding received messages to appropriate channels. Program O employs a table (*PID-Channel*) to map process IDs to in/out channels (Figure 17b). Therefore, it can check if the process ID in a request belongs to the sender. Furthermore, O forwards received responses to processes.

Program O commits to the intermediate controller that the PID in a request belongs to the sender and is unique inside the node by performing the first task and checking requests against the table. Furthermore, it commits to processes that they will receive their valid responses using the table.

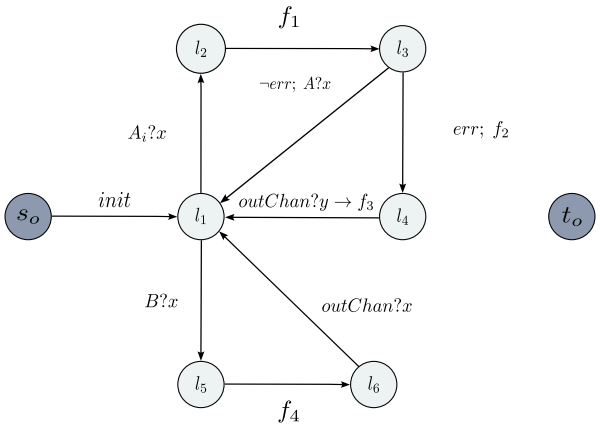


Figure 19: Program of the Operating System.

Program O receives a resource-allocation request from process p_i via channel A_i and checks the request. If the check passes, the program forwards the request toward program I through channel A. Otherwise, it returns an error message to program P_i. In addition, It receives the response from program I via channel B, finds the appropriate channel, and sends the response to program p_j through this channel. Figure 19 depicts the state diagram of program O. The conditions and functions of program O are as follows:

$$
\begin{aligned}
init(\sigma) &= (\sigma : tablePtr \mapsto getTablePtr(\sigma(OSTable))),\\
f_1(\sigma) &= (\sigma : err \mapsto checkIsUniquePID(\sigma(x), tablePtr)),\\
f_2(\sigma) &= (\sigma : y, outChan \mapsto createErrResponse(\sigma(x))),\\
f_3(\sigma) &= (\sigma : err \mapsto false),\\
f_4(\sigma) &= (\sigma : outChan, \mapsto findChannel(\sigma(x), tablePtr)),
\end{aligned}
$$

where $OSTable$ is the $PID - channel$ table of the operating system

The A-C formula for process O is as follows:

$$\vdash \langle A^o,\ C^o \rangle :\ \{\varphi^o\}\ O\ \{\psi^o\}$$

where the formal definition of φ^o, ψ^o , A^o, and C^o are as follows:

$$
\begin{aligned}
\varphi^o &\stackrel{\text{def}}{=} \#A = \#B = 0\ \wedge\ \neg err\ \wedge\ \#outChan = 0\\
\psi^o &\stackrel{\text{def}}{=} false\\
A^o &\stackrel{\text{def}}{=} \#B > 0 \rightarrow isValidCap(last(B).cap)\\
C^o &\stackrel{\text{def}}{=} (\#A > 0 \rightarrow\ isUniquePID(last(A).spid)) \wedge
\end{aligned}
$$

$$((\#outChan > 0 \ \wedge \ \neg err) \rightarrow \ isValidCap(last(outChan).cap))$$

Where *outChannel* is a logical variable.
The assertion network for O is as follows:

$$Q_{s_o} \stackrel{\text{def}}{=} \#A = \#B = 0 \ \wedge \ \neg err \ \wedge \ \#outChan = 0$$

$$Q_1 \stackrel{\text{def}}{=} \neg err$$

$$Q_2 \stackrel{\text{def}}{=} \neg err \ \wedge \ last(A_i.spid) = x$$

$$Q_3 \stackrel{\text{def}}{=} last(A_i.spid) = x$$

$$Q_4 \stackrel{\text{def}}{=} err$$

$$Q_5 \stackrel{\text{def}}{=} last(B) = x \ \wedge \ \neg err \ \wedge \ isValidCap(last(B).cap)$$

$$Q_6 \stackrel{\text{def}}{=} last(B) = x \ \wedge \ isValidCap(x.cap)) \ \wedge \ \neg err$$

$$Q_{t_o} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption A^o, commitment C^o, and channels A, B, A_i, B_i such that $1 \leqslant i \leqslant n$.

- $\models Q_{s_o} \rightarrow C^o$ follows from the above definitions.
- $\models Q_{s_o} \wedge A^o \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* just assign a pointer to the table to variable *tablePtr* and $\sigma \models \neg err$. Hence, this verification holds.
- $\models Q_{l_1} \wedge A^o \rightarrow ((A^o \rightarrow Q_{l_2}) \wedge C^o) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(A_i, v))$ for arbitrary v. In this input transition, a communication through channel A_i, such that $1 \leqslant i \geqslant n$, just took place. Function g assigns the last received value from channel A_i to x. Thus, this verification holds.
- $\models Q_{l_2} \ \wedge \ A^o \rightarrow (A^o \rightarrow Q_{l_3}) \circ f_1$. In this internal transition, function f_1 checks the request to see if it is unique in the node and assign the result to variable *err* Thus, this verification holds.
- $\models Q_{l_3} \wedge \neg err \wedge A^o \rightarrow ((A^o \rightarrow Q_{l_1}) \wedge C^o) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A, \sigma(x)))$. In this output transition, program O just sends out variable x through channel A. The condition of the transition states that there is no error, which means the process ID in the request is unique. Therefore, commitment C^o is satisfied. Hence, this verification holds.
- $\models Q_{l_3} \ \wedge \ err \ \wedge \ A^o \rightarrow (A^o \rightarrow Q_{l_4}) \circ f_2$. In this internal transition, the condition of the transition states that there is error. Furthermore, the commitment C^o is satisfied because $\sigma \models err$. Thus, this verification holds.
- $\models \ Q_{l_4} \ \wedge \ A^o \ \rightarrow \ ((A^o \ \rightarrow \ Q_{l_1}) \ \wedge \ C^o) \circ (f_3 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma \ : \ h \ \mapsto \ \sigma(h).(outChan, \sigma(y)))$. In this output transition, program O just sends out variable y through channel *outChan*. Because $\sigma \models err$, commitment C^o is satisfied. In addition, function f_1 does unset variable *err*. Hence, this verification holds.

- $\models Q_{l_1} \wedge A^o \rightarrow ((A^o \rightarrow Q_{l_5}) \wedge C^o) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(B, v))$ for arbitrary v. In this input transition, a communication through channel B just took place and function g assigns the last received value to x. $\sigma' \models isValidCap(last(B).cap)$ Because $\#B > 0$, Thus, this verification holds.

- $\models Q_{l_5} \wedge A^o \rightarrow (A^o \rightarrow Q_{l_6}) \circ f_4$. In this internal transition, function f_4 finds the appropriate output channel and assigns it to *outChan*. Furthermore, x contains a valid capability because it contains the last received message from channel B and $\sigma \models isValidCap(last(B).cap)$. Hence, this verification holds.

- $\models Q_{l_6} \wedge A^o \rightarrow ((A^o \rightarrow Q_{l_1}) \wedge C^o) \circ (f_3 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(outChan, \sigma(x)))$. In this output transition, program O just sends out variable y through channel *outChan*. Because $\sigma \models \neg err \wedge isValidCap(x.cap)$, commitment C^o is satisfied. Thus, this verification holds.

Since $Q_{t^o} \rightarrow \psi^o$, the post-condition of program O is satisfied.

6.2.2 Intermediate Controller

We follow the pattern in which we reasoned about the resource controller to reason about the intermediate controller regarding the parallel composition rule. We model the intermediate controller by program I as the parallel execution of the above programs as follows:

$$I \stackrel{\text{def}}{=} I_{qh} \parallel I^q_{ra} \parallel I_{id} \parallel I^q_{ed} \parallel I_{ir} \parallel I^q_{er} \parallel I_{sh} \parallel I^s_{ra} \parallel I^s_{ed} \parallel I^s_{er}$$

With the A-C formula $\vdash \langle A^i, C^i \rangle : \{\varphi^i\}\ I\ \{\psi^i\}$, after applying the parallel composition rule. The commitments of the intermediate controller will be the combination of its programs. Assumption A^i indicates that the intermediate controller assumes the process ID inside each request is unique in the PE node, and responses from resource controllers contain valid capabilities. However, before checking the correctness of its assumptions, we describe how they cooperate to handle requests and responses.

The intermediate controller handles requests and responses in parallel via request and response handlers Each handler has three tasks: locking the capability tree, forwarding its input to the appropriate module, and sending the response of a module to the output channel. Parallel handling of requests and responses may cause manipulating the tree simultaneously, thus,breaking the tree's well-formedness The handlers employ a locking mechanism ($\mathscr{L}$) to lock the tree before processing each input to prevent the situation. We assume $\mathscr{L}$ performs the locking task correctly and fairly. Therefore, controllers can always guarantee the tree's well-formedness. We now explain how each handler accomplishes its task.

6.2.2.1 Request Handling

We model the request handler by the program I_{qh}. Program I_{qh} cooperates with five other programs to handle requests, including I^q_{ra} (resource-allocation request handler),

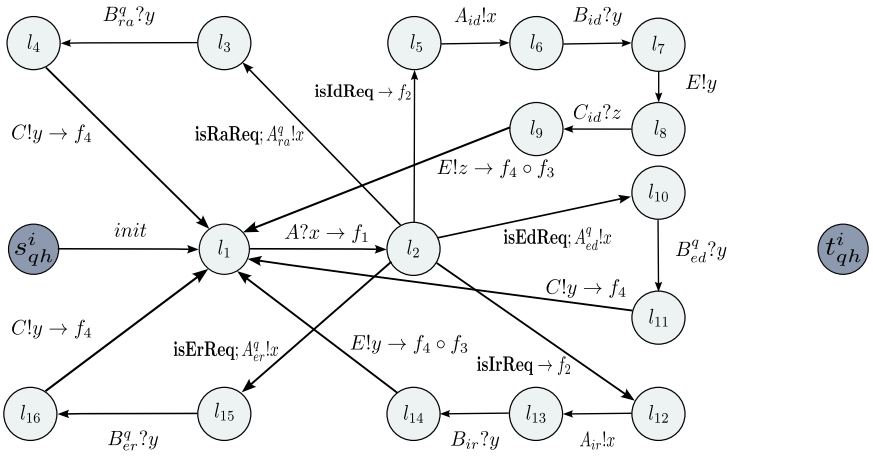


Figure 20: Request-Handler Program of the Intermediate Controller.

I_{id} (the internal-delegation handler), I^q_{ed} (the external-delegation request handler), I_{ir} (the internal-revocation handler), and I^q_{er} (the external-revocation request handler). Program I_{qh} assumes that the PID in a request belongs to the sender and is unique inside the node. I_{qh} commits to forward the request to the corresponding program and forwarding programs' responses via output channels. Figure 20 depicts the state diagram of program I_{qh}.

Program I_{qh} receives requests of the processes inside a process node via channel A. It uses five predicates to find out the type of a request and forward the request to the corresponding module. Then, the handler receives the response and forwards it to the appropriate channel. If the request is handled locally, the handler forwards the response to channel C_{12}. Otherwise, the handler forwards the response of the module to a remote resource node via channel B. The conditions and functions of I_{qh} are defined as follow:

$$
\begin{aligned}
init(\sigma) = \; & (\sigma : rootPtr, treeLocked, \\
& isRaReq, isIdReq, isEdReq, isIrReq, isErReq \mapsto \\
& getRoot(\sigma(icTree)), \; false, \; false, \; false, \; false, \; false, \; false), \\
f_1(\sigma) \; = \; & (\sigma : isRaReq, isIdReq, isEdReq, isIrReq, isErReq \mapsto \\
& isRaReqCheck(\sigma(x)), isIdReqCheck(\sigma(x)), isEdReqCheck(\sigma(x)), \\
& isIrReqCheck(\sigma(x)), isErReqCheck(\sigma(x))), \\
f_2(\sigma) \; = \; & (\sigma : treeLocked \mapsto lockTree(\sigma(rootPtr))), \\
f_3(\sigma) \; = \; & (\sigma : treeLocked \mapsto unlockTree(\sigma(rootPtr))), \\
f_4(\sigma) \; = \; & (\sigma : isRaReq, isIdReq, isEdReq, isIrReq, isErReq \mapsto \\
& false, \; false, \; false, \; false, \; false)
\end{aligned}
$$

The A-C formula for process I_{qh} is as follows:

$$\vdash \langle A^i_{qh}\ C^i_{qh}\rangle\ \{\varphi^i_{qh}\}\ I_{qh}\ \{\psi^i_{qh}\}$$

where the formal definition of φ^i_{qh}, ψ^i_{qh} , A^i_{qh}, and C^i_{qh} are as follows:

$$\begin{aligned}
\varphi^i_{qh} &\stackrel{\text{def}}{=} \#A = \#C = \#E = 0 \ \wedge\ \#A^q_{ra} = \#B^q_{ra} = 0 \ \wedge\ \#A_{id} = \#B_{id} = \#C_{id} = 0 \wedge \\
&\quad \#A^q_{ed} = \#B^q_{ed} = 0 \ \wedge\ \#A_{ir} = \#B_{ir} = 0 \ \wedge\ \#A^q_{er} = \#B^q_{er} = 0 \\
\psi^i_{qh} &\stackrel{\text{def}}{=} false \\
A^i_{qh} &\stackrel{\text{def}}{=} (\#A > 0 \rightarrow isUniquePID(last(A).spid)) \\
&\quad ((\#B^q_{ra} > 0) \rightarrow isUniquePID(last(B^q_{ra}).spid)) \\
&\quad ((\#B_{id} > 0) \rightarrow isValidCap(last(B_{id}).cap)) \wedge \\
&\quad ((\#C_{id} > 0) \rightarrow isIdConfRes(last(C_{id}).cap)) \wedge \\
&\quad ((\#B^q_{ed} > 0) \rightarrow isUniquePID(last(B^q_{ed}).spid)) \wedge \\
&\quad ((\#B_{ir} > 0) \rightarrow isIrConfRes(last(B_{ir}))) \wedge \\
&\quad ((\#B^q_{er} > 0) \rightarrow isUniquePID(last(B^q_{er}).spid)) \\
C^i_{qh} &\stackrel{\text{def}}{=} (\#C > 0) \rightarrow isUniquePID(last(C).spid)) \wedge \\
&\quad (\#E > 0 \ \wedge\ isIrReq \rightarrow isIrConfRes(last(E))) \wedge \\
&\quad (\#E > 0 \ \wedge\ \neg isIrReq) \rightarrow isValidCap(last(E).cap)) \wedge \\
&\quad ((\#A^q_{ra} > 0 \ \wedge\ \#A_{ra} \neq \#B_{ra}) \rightarrow isUniquePID(last(A^q_{ra}).spid) \wedge \\
&\quad isRaReq(last(A^q_{ra})) \wedge \\
&\quad ((\#A_{id} > 0) \rightarrow isUniquePID(last(A_{id}).spid)) \wedge \\
&\quad isIdReq(last(A_{ir})) \wedge \\
&\quad ((\#A^q_{ed} > 0) \rightarrow isUniquePID(last(A^q_{ed}).spid)) \wedge \\
&\quad isEdReq(last(A^q_{ed})) \wedge \\
&\quad ((\#A_{ir} > 0) \rightarrow isUniquePID(last(A_{ir}).spid)) \wedge \\
&\quad isIrReq(last(A_{ir})) \wedge \\
&\quad ((\#A^q_{er} > 0) \rightarrow isUniquePID(last(A^q_{er}).spid)) \\
&\quad isErReq(last(A^q_{er}))
\end{aligned}$$

The assertion network for I_{qh} is as follows:

$$\begin{aligned}
Q_{s^i_{qh}} &\stackrel{\text{def}}{=} \#A = \#C = \#E = 0 \ \wedge\ \#A^q_{ra} = \#B^q_{ra} = 0 \ \wedge\ \#A_{id} = \#B_{id} = \#C_{id} = 0 \wedge \\
&\quad \#A^q_{ed} = \#B^q_{ed} = 0 \ \wedge\ \#A_{ir} = \#B_{ir} = 0 \ \wedge\ \#A^q_{er} = \#B^q_{er} = 0 \\
Q_1 &\stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge\ \#A_{id} = \#B_{id} = \#C_{id} \ \wedge\ \#A^q_{ed} = \#B^q_{ed} \ \wedge\ \#A_{ir} = \#B_{ir} \wedge \\
&\quad \#A^q_{er} = \#B^q_{er} \ \wedge\ \neg treeLocked \wedge \\
&\quad \neg isRaReq \ \wedge\ \neg isIdReq \ \wedge\ \neg isEdReq \ \wedge\ \neg isIrReq \ \wedge\ \neg isErReq \\
Q_2 &\stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge\ \#A_{id} = \#B_{id} = \#C_{id} \ \wedge\ \#A^q_{ed} = \#B^q_{ed} \ \wedge\ \#A_{ir} = \#B_{ir} \wedge
\end{aligned}$$

$$\#A^q_{er} = \#B^q_{er} \ \wedge \ \neg treeLocked \ \wedge \ last(A) = x \ \wedge \ isUniquePID(last(A).spid)$$

$$Q_3 \stackrel{\text{def}}{=} (\#A^q_{ra} - 1) = \#B^q_{ra} \ \wedge \ \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ \neg treeLocked \ \wedge \ last(A) = x \ \wedge \ last(A^q_{ra}) = x \ \wedge$$
$$isRaReq \ \wedge \ \neg isIdReq \ \wedge \ \neg isEdReq \ \wedge \ \neg isIrReq \ \wedge \ \neg isErReq$$

$$Q_4 \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ \neg treeLocked \ \wedge \ last(A) = x \ \wedge$$
$$last(A^q_{ra}) = x \ \wedge \ last(B^q_{ra}) = y \ \wedge \ isUniquePID(last(B^q_{ra}).spid)$$
$$isRaReq \ \wedge \ \neg isIdReq \ \wedge \ \neg isEdReq \ \wedge \ \neg isIrReq \ \wedge \ \neg isErReq$$

$$Q_5 \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ treeLocked \ \wedge \ last(A) = x \ \wedge \ isUniquePID(last(A).spid) \ \wedge$$
$$\neg isRaReq \ \wedge \ isIdReq \ \wedge \ \neg isEdReq \ \wedge \ \neg isIrReq \ \wedge \ \neg isErReq$$

$$Q_6 \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ (\#A_{id} - 1) = \#B_{id} = \#C_{id} \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ treeLocked \ \wedge \ last(A) = x \ \wedge \ last(A_{id}) = x \ \wedge$$
$$\neg isRaReq \ \wedge \ isIdReq \ \wedge \ \neg isEdReq \ \wedge \ \neg isIrReq \ \wedge \ \neg isErReq$$

$$Q_7 \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ \#A_{id} = \#B_{id} = (\#C_{id} + 1) \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ treeLocked \ \wedge \ last(A) = x \ \wedge \ last(A_{id}) = x \ \wedge$$
$$last(B_{id}) = y \ \wedge \ isValidCap(last(B_{id}).cap) \ \wedge$$
$$\neg isRaReq \ \wedge \ isIdReq \ \wedge \ \neg isEdReq \ \wedge \ \neg isIrReq \ \wedge \ \neg isErReq$$

$$Q_8 \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ \#A_{id} = \#B_{id} = (\#C_{id} + 1) \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ treeLocked \ \wedge \ last(A) = x \ \wedge \ last(A_{id}) = x \ \wedge \ last(B_{id}) = y \ \wedge$$
$$\neg isRaReq \ \wedge \ isIdReq \ \wedge \ \neg isEdReq \ \wedge \ \neg isIrReq \ \wedge \ \neg isErReq$$

$$Q_9 \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ \#A_{id} = \#B_{id} = (\#C_{id} + 1) \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ treeLocked \ \wedge \ last(A) = x \ \wedge \ last(A_{id}) = x \ \wedge \ last(B_{id}) = y \ \wedge$$
$$last(C_{id}) = z \ \wedge \ isValidCap(last(C_{id}).cap) \ \wedge$$
$$\neg isRaReq \ \wedge \ isIdReq \ \wedge \ \neg isEdReq \ \wedge \ \neg isIrReq \ \wedge \ \neg isErReq$$

$$Q_{10} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ (\#A^q_{ed} - 1) = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ \neg treeLocked \ \wedge \ last(A) = x \ \wedge \ last(A^q_{ed}) = x \ \wedge$$
$$\neg isRaReq \ \wedge \ \neg isIdReq \ \wedge \ isEdReq \ \wedge \ \neg isIrReq \ \wedge \ \neg isErReq$$

$$Q_{11} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ \neg treeLocked \ \wedge \ last(A) = x \ \wedge$$
$$last(A^q_{ed}) = x \ \wedge \ last(B^q_{ed}) = y \ \wedge \ isUniquePID(last(B^q_{ed}).spid)$$
$$\neg isRaReq \ \wedge \ \neg isIdReq \ \wedge \ isEdReq \ \wedge \ \neg isIrReq \ \wedge \ \neg isErReq$$

$$Q_{12} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \ \wedge \ \#A_{ir} = \#B_{ir} \ \wedge$$
$$\#A^q_{er} = \#B^q_{er} \ \wedge \ treeLocked \ \wedge \ last(A) = x \ \wedge \ isUniquePID(last(A).spid) \ \wedge$$

$$\neg isRaReq \wedge \neg isIdReq \wedge \neg isEdReq \wedge isIrReq \wedge \neg isErReq$$

$$Q_{13} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \wedge \#A_{id} = \#B_{id} = \#C_{id} \wedge \#A^q_{ed} = \#B^q_{ed} \wedge (\#A_{ir} - 1) = \#B_{ir} \wedge \#A^q_{er} = \#B^q_{er} \wedge treeLocked \wedge last(A) = x \wedge last(A_{ir}) = x \wedge \neg isRaReq \wedge \neg isIdReq \wedge \neg isEdReq \wedge isIrReq \wedge \neg isErReq$$

$$Q_{14} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \wedge \#A_{id} = \#B_{id} = \#C_{id} \wedge \#A^q_{ed} = \#B^q_{ed} \wedge \#A_{ir} = \#B_{ir} \wedge \#A^q_{er} = \#B^q_{er} \wedge treeLocked \wedge last(A) = x \wedge last(A_{ir}) = x \wedge last(B_{ir}) = y \wedge isIrConfRes(last(B_{ir})) \wedge \neg isRaReq \wedge \neg isIdReq \wedge \neg isEdReq \wedge isIrReq \wedge \neg isErReq$$

$$Q_{15} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \wedge \#A_{id} = \#B_{id} = \#C_{id} \wedge \#A^q_{ed} = \#B^q_{ed} \wedge \#A_{ir} = \#B_{ir} \wedge (\#A^q_{er} - 1) = \#B^q_{er} \wedge \neg treeLocked \wedge last(A) = x \wedge last(A^q_{ed}) = x \wedge \neg isRaReq \wedge \neg isIdReq \wedge \neg isEdReq \wedge \neg isIrReq \wedge isErReq$$

$$Q_{16} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \wedge \#A_{id} = \#B_{id} = \#C_{id} \wedge \#A^q_{ed} = \#B^q_{ed} \wedge \#A_{ir} = \#B_{ir} \wedge \#A^q_{er} = \#B^q_{er} \wedge \neg treeLocked \wedge last(A) = x \wedge last(A^q_{er}) = x \wedge last(B^q_{er}) = y \wedge isUniquePID(last(B^q_{er}).spid) \wedge \neg isRaReq \wedge \neg isIdReq \wedge \neg isEdReq \wedge \neg isIrReq \wedge isErReq$$

$$Q_{t^i_{qh}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption A^i_{qh}, commitment C^i_{qh}, and channels A, C, E,A^q_{ra}, B^q_{ra}, A_{id}, B_{id}, A^q_{ed}, B^q_{ed}, A_{ir}, B_{ir}, A^q_{er}, and B^q_{er}.

- $\models Q_{s^i_{qh}} \rightarrow C^i_{qh}$ follows from the above definitions.
- $\models Q_{s^i_{qh}} \wedge A^i_{qh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does unset boolean variables *treeLocked*, *isRaReq*, *isIdReq*, *isIrReq*, *isEdReq*, and *isErReq*. Hence, this verification holds.
- $\models Q_{l_1} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_2}) \wedge C^i_{qh} \circ (f_1 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(A, v))$ for arbitrary v. In this input transition, a communication through channel A just took place. Function g assigns the last received value from channel A to x. The first implication of A^i_{qh} satisfies $isUniquePID(last(A).spid)$ predicate because $\#A > 0$. Function f_1 checks the variable x to find the request type and assigns boolean variables *isRaReq*, *isIdReq*, *isIrReq*, *isEdReq*, and *isErReq*, accordingly. Thus, this verification holds.
- $\models Q_{l_2} \wedge isRaReq \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_3}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ra}, \sigma(x)))$. In this output transition, program I_{qh} sends x through channel A^q_{ra} toward program I^q_{ra}. From the transition's condition we have *isRaReq*. C^i_{qh} holds because $\#A^q_{ra} > 0$ and $\sigma \models isUniquePID(x.spid)$. Hence, this verification holds.
- $\models Q_{l_3} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_4}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B^q_{ra}, v))$ for arbitrary v. In this input transition, a communication through channel B^q_{ra} just

took place and function g assigns the last received value to y. Because $\#A^q_{ra} = \#B^q_{ra} > 0$, the second implication of A^i_{qh} satisfies $isUniquePID(last(B^q_{ra}).spid)$. Thus, this verification holds.

- $\models Q_{l_4} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_1}) \wedge C^i_{qh}) \circ (f_4 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y)))$. In this output transition, program I_{qh} just sends out variable y through channel C. Furthermore, function f_4 does unset boolean variables $isRaReq$, $isIdReq$, $isIrReq$, $isEdReq$, and $isErReq$. C^i_{qh} holds because $\#C > 0$, and $\sigma \models isUniquePID(last(C).cap)$. Hence, this verification holds.

- $\models Q_{l_2} \wedge isIdReq \wedge A^i_{qh} \rightarrow Q_{l_{i,5}} \circ f_1$. In this internal transition, function f_1 locks tree and set the variable $lockedTree$. Furthermore, the condition of the transition satisfies that $isIrReg$ is set. Thus, this verification holds.

- $\models Q_{l_5} \wedge isidReq \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_6}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{id}, \sigma(x)))$. In this output transition, program I_{qh} sends x through channel A_{id} toward program I_{id}. C^i_{qh} holds because $\#A_{id} > 0$ and from Q_{l_5} we have $isUniquePID(x.spid)$. Hence, this verification holds.

- $\models Q_{l_6} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_7}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B_{id}, v))$ for arbitrary v. In this input transition, a communication through channel B_{id} just took place and function g assigns the last received value to y. Because $\#B_{id} > 0$, the second implication of A^i_{qh} satisfies $isValidCap(last(B_{id}).cap)$. Thus, this verification holds.

- $\models Q_{l_7} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_8}) \wedge C^i_{qh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(E, \sigma(y)))$. In this output transition, program I_{qh} just sends out variable y through channel E. C^i_{qh} holds because $\#E > 0$ and $\sigma \models valid(last(C).cap)$. Hence, this verification holds.

- $\models Q_{l_8} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_9}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z, h \mapsto v, \sigma(h).(C_{id}, v))$ for arbitrary v. In this input transition, a communication through channel C_{id} just took place and function g assigns the last received value to z. Because $\#C_{id} > 0$, the second implication of A^i_{qh} satisfies $isIdConfRes(last(C_{id}))$. Thus, this verification holds.

- $\models Q_{l_9} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_1}) \wedge C^i_{qh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(E, \sigma(z)))$. In this output transition, program I_{qh} just sends out variable z through channel E. Because $\#E > 0 \wedge isIrReq$, then C^i_{qh} holds. Furthermore, function f_4 does unset boolean variables $treeLocked$, $isRaReq$, $isIdReq$, $isIrReq$, $isEdReq$, and $isErReq$. Hence, this verification holds.

- $\models Q_{l_2} \wedge isEdReq \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_{10}}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ed}, \sigma(x)))$. In this output transition, program I_{qh} sends x through channel A^q_{ed} toward program I^q_{ed}. From the transition's condition we have $isEdReq$. C^i_{qh} holds because $\#A^q_{ed} > 0$ and $\sigma \models isUniquePID(x.spid)$. Hence, this verification holds.

- $\models Q_{l_{10}} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_{11}}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B^q_{ed}, v))$ for arbitrary v. In this input transition, a communication through channel B^q_{ed} just took place and function g assigns the last received value to y. the implication of A^i_{qh} satisfies $isUniquePID(last(B^q_{ed}).spid)$ because $\#A^q_{ed} = \#B^q_{ed} > 0$. Thus, this verification holds.

- $\models Q_{l_{11}} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_1}) \wedge C^i_{qh}) \circ (f_4 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y)))$. In this output transition, program I_{qh} just sends out variable y through channel C. Furthermore, function f_4 does unset boolean variables $isRaReq$, $isIdReq$, $isIrReq$, $isEdReq$, and $isErReq$. C^i_{qh} holds because $\#C > 0$, and $\sigma \models isUniquePID(last(C).cap)$. Hence, this verification holds.

- $\models Q_{l_2} \wedge isIrReq \wedge A^i_{qh} \rightarrow Q_{l_{i,12}} \circ f_1$. In this internal transition, function f_1 locks tree and set the variable $lockedTree$. Furthermore, the condition of the transition satisfies that $isIrReg$ is unset. Thus, this verification holds.

- $\models Q_{l_{12}} \wedge isidReq \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_{13}}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{ir}, \sigma(x)))$. In this output transition, program I_{qh} sends x through channel A_{ir} toward program I_{ir}. C^i_{qh} holds because $\#A_{ir} > 0$ and $\sigma \models isUniquePID(x.spid)$. Hence, this verification holds.

- $\models Q_{l_{13}} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_{14}}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B_{ir}, v))$ for arbitrary v. In this input transition, a communication through channel B_{ir} just took place and function g assigns the last received value to y. Because $\#B_{ir} > 0$, the second implication of A^i_{qh} satisfies $isIrConfRes(last(B_{ir}).cap)$. Thus, this verification holds.

- $\models Q_{l_{14}} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_1}) \wedge C^i_{qh}) \circ (f_4 \circ f_3 \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(E, \sigma(y)))$. In this output transition, program I_{qh} just sends out variable y through channel E. C^i_{qh} holds because $\#E > 0$ and $\sigma \models isIrConfRes(last(E).cap)$. Function f_3 unlocks the tree and function f_4 does unset boolean variables $isRaReq$, $isIdReq$, $isIrReq$, $isEdReq$, and $isErReq$. Hence, this verification holds.

- $\models Q_{l_2} \wedge isErReq \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_{15}}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{er}, \sigma(x)))$. In this output transition, program I_{qh} sends x through channel A^q_{er} toward program I^q_{er}. From the transition's condition we have $isErReq$. C^i_{qh} holds because $\#A^q_{er} > 0$ and $\sigma \models isUniquePID(x.spid)$. Hence, this verification holds.

- $\models Q_{l_{15}} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_{16}}) \wedge C^i_{qh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B^q_{er}, v))$ for arbitrary v. In this input transition, a communication through channel B^q_{ed} just took place and function g assigns the last received value to y. A^i_{qh} satisfies $isUniquePID(last(B^q_{er}).spid)$ Because $\#A^q_{er} = \#B^q_{er} > 0$. Thus, this verification holds.

- $\models Q_{l_{16}} \wedge A^i_{qh} \rightarrow ((A^i_{qh} \rightarrow Q_{l_1}) \wedge C^i_{qh}) \circ (f_4 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y)))$. In this output transition, program I_{qh} just sends out variable y through channel C. Furthermore, function f_4 does unset boolean variables $isRaReq$,

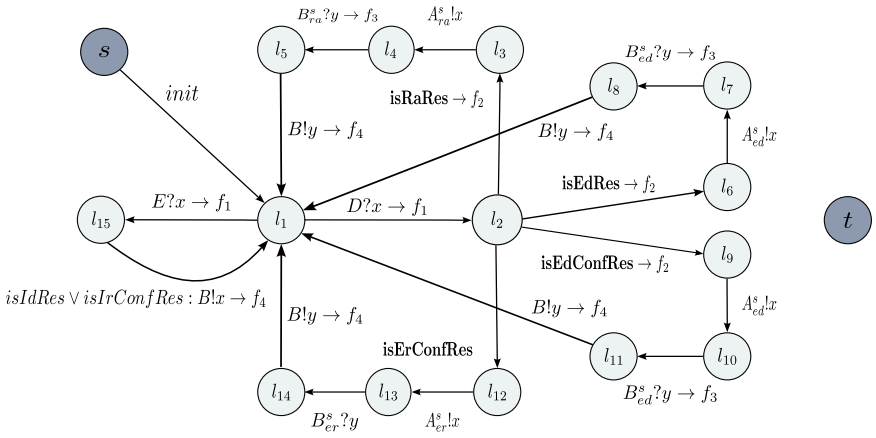


Figure 21: Response-Handler Program of the Intermediate Controller.

isIdReq, *isIrReq*, *isEdReq*, and *isErReq*. C^i_{qh} holds because $\#C > 0$, and $\sigma \models isUniquePID(last(C).cap)$. Hence, this verification holds.

Since $Q_{t^i_{qh}} \to \psi^i_{qh}$, the post-condition of program I_{qh} is satisfied.

6.2.2.2 Response Handling

We model the response handler by the program I_{sh}. It cooperates with three programs to handle responses, including I^s_{ra} (resource-allocation response handler), I^s_{ed} (the external-delegation response handler), and I^s_{er} (the external-revocation response handler). In addition, I_{sh} directly forwards all the responses from the request handler to O. Program I_{sh} assumes that all received responses from the resource controllers are valid. Furthermore, it commits to forwarding received responses to the corresponding programs and forwarding programs' responses to O directly. Figure 21 depicts the state diagram of program I_{sh}.

Program I_{sh} receives responses from resource controllers via channel D. It uses four predicates to find out the type of a responses and forward it to the corresponding module. Then, the handler receives the response and forwards it to the operating system via channel B. In addition, it receives the responses from the request handler through channel E and forward them via channel B The conditions and functions of I_{sh} are defined as follow:

$$
\begin{aligned}
init(\sigma) = \ & (\sigma : rootPtr, treeLocked, \\
& isRaRes, isEdRes, isEdConfRes, isErConfRes, isIdfRes, isIrConfRes \\
& \mapsto getRoot(\sigma(icTree)), false, false, false, false, false\ false), \\
f_1(\sigma) \ = \ & (\sigma : isRaRes, isEdRes, isEdConfRes, isErConfRes, isIdfRes, \\
& isIrConfRes \mapsto isRaResCheck(\sigma(x)), isEdResCheck(\sigma(x)),
\end{aligned}
$$

$$
\begin{aligned}
& isEdConfResCheck(\sigma(x)),\ isErConfResCheck(\sigma(x)),\\
& isIdfResCheck(\sigma(x)),\ isIrConfRes(\sigma(x))),\\
f_2(\sigma) &= (\sigma : treeLocked \mapsto lockTree(\sigma(rootPtr))),\\
f_3(\sigma) &= (\sigma : treeLocked \mapsto unlockTree(\sigma(rootPtr))),\\
f_4(\sigma) &= (\sigma : isRaRes,\ isEdRes,\ isEdConfRes,\ isErConfRes,\ isIdfRes,\\
& \quad isIrConfRes \mapsto false,\ false,\ false,\ false,\ false,\ false)
\end{aligned}
$$

The A-C formula for process I_{sh} is as follows:

$$\vdash \langle A^i_{sh}\ C^i_{sh}\rangle\ \{\varphi^i_{sh}\}\ I_{sh}\ \{\psi^i_{sh}\}$$

where the formal definition of φ^i_{sh}, ψ^i_{sh} , A^i_{sh}, and C^i_{sh} are as follows:

$$
\begin{aligned}
\varphi^i_{sh} &\stackrel{\text{def}}{=} \#B = \#D = \#E = 0 \ \wedge\ \#A^s_{ra} = \#B^s_{ra} = 0 \ \wedge\ \#A^s_{ed} = \#B^s_{ed} = 0 \wedge\\
&\quad \#A^s_{er} = \#B^s_{er} = 0 \ \wedge\ \neg treeLocked \ \wedge\ \neg isRaRes \ \wedge\ \neg isEdRes \wedge\\
&\quad \neg isEdConfRes \ \wedge\ \neg isIdfRes \ \wedge\ \neg isIrConfRes\\
\psi^i_{sh} &\stackrel{\text{def}}{=} false\\
A^i_{sh} &\stackrel{\text{def}}{=} ((\#D > 0 \wedge (isRaRes \vee isEdRes \vee isEdConfRes))\\
&\qquad \rightarrow isValidCap(last(D).cap)) \wedge\\
&\quad ((\#B^s_{ra} > 0) \rightarrow isValidCap(last(B^s_{ra}).cap)) \wedge\\
&\quad ((\#B^s_{ed} > 0) \rightarrow isValidCap(last(B^s_{ed}).cap))\\
&\quad ((\#B^s_{er} > 0) \rightarrow isErConfRes(last(B^s_{er}))) \wedge\\
C^i_{sh} &\stackrel{\text{def}}{=} ((\#B > 0 \wedge (isRaRes \vee isEdRes \vee isEdConfRes \vee isIdfRes))\\
&\qquad \rightarrow isValidCap(last(B).cap))) \wedge\\
&\quad ((\#B > 0 \wedge isErConfRes) \rightarrow isErConfRes(last(B)))) \wedge\\
&\quad ((\#A^s_{ra} > 0) \rightarrow isRaRes(last(A^s_{ra}))\ \wedge\ isValidCap(last(A^s_{ra}).cap)) \wedge\\
&\quad ((\#A^s_{ed} > 0) \rightarrow (isEdRes(last(A^s_{ed})) \vee isEdConfRes(last(A^s_{ed})) \wedge\\
&\qquad\qquad isValidCap(last(A^s_{ed}).cap))
\end{aligned}
$$

The assertion network for I_{sh} is as follows:

$$
\begin{aligned}
Q_{s^i_{sh}} &\stackrel{\text{def}}{=} \#B = \#D = \#E = 0 \ \wedge\ \#A^s_{ra} = \#B^s_{ra} = 0 \ \wedge\ \#A^s_{ed} = \#B^s_{ed} = 0 \wedge\\
&\quad \#A^s_{er} = \#B^s_{er} = 0 \ \wedge\ \neg treeLocked \ \wedge\ \neg isRaRes \ \wedge\ \neg isEdRes \wedge\\
&\quad \neg isEdConfRes \ \wedge\ \neg isErConfRes \ \wedge\ \neg isIdfRes \ \wedge\ \neg isIrConfRes\\
Q_2 &\stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} = 0 \ \wedge\ \#A^s_{ed} = \#B^s_{ed} = 0 \ \wedge\ \#A^s_{er} = \#B^s_{er} = 0 \ \wedge\ \neg treeLocked \wedge\\
&\quad last(D) = x\\
Q_3 &\stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} = 0 \ \wedge\ \#A^s_{ed} = \#B^s_{ed} = 0 \ \wedge\ \#A^s_{er} = \#B^s_{er} = 0 \ \wedge\ last(D) = x\\
&\quad treeLocked \ \wedge\ isRaRes \ \wedge\ \neg isEdRes \ \wedge\ \neg isEdConfRes \wedge
\end{aligned}
$$

$$\neg isErConfRes \wedge negisIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge isValidCap(last(D))$$

$$Q_4 \stackrel{\text{def}}{=} \#A^s_{ra} - 1 = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x \; treeLocked \wedge isRaRes \wedge \neg isEdRes \wedge \neg isEdConfRes \wedge \neg isErConfRes \wedge \neg isIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge last(A^s_{ra}) = x \wedge isValidCap(last(A^s_{ra}))$$

$$Q_5 \stackrel{\text{def}}{=} \#A^s_{ra} - 1 = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x \; \neg treeLocked \wedge isRaRes \wedge \neg isEdRes \wedge \neg isEdConfRes \wedge \neg isErConfRes \wedge \neg isIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge last(A^s_{ra}) = x \wedge last(B^s_{ra}) = y \wedge isValidCap(last(B^s_{ra}))$$

$$Q_6 \stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x \; treeLocked \wedge \neg isRaRes \wedge isEdRes \wedge \neg isEdConfRes \wedge \neg isErConfRes \wedge negisIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge isValidCap(last(D))$$

$$Q_7 \stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} = 0 \wedge \#A^s_{ed} - 1 = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x \; treeLocked \wedge \neg isRaRes \wedge isEdRes \wedge \neg isEdConfRes \wedge \neg isErConfRes \wedge \neg isIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge last(A^s_{ra}) = x \wedge isValidCap(last(A^s_{ed}))$$

$$Q_8 \stackrel{\text{def}}{=} \#A^s_{ra} - 1 = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x \; \neg treeLocked \wedge \neg isRaRes \wedge isEdRes \wedge \neg isEdConfRes \wedge \neg isErConfRes \wedge \neg isIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge last(A^s_{ed}) = x \wedge last(B^s_{ed}) = y \wedge isValidCap(last(B^s_{ed}))$$

$$Q_9 \stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x \; treeLocked \wedge \neg isRaRes \wedge \neg isEdRes \wedge isEdConfRes \wedge \neg isErConfRes \wedge negisIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge isValidCap(last(D))$$

$$Q_{10} \stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} = 0 \wedge \#A^s_{ed} - 1 = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x \; treeLocked \wedge \neg isRaRes \wedge \neg isEdRes \wedge isEdConfRes \wedge \neg isErConfRes \wedge \neg isIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge last(A^s_{ra}) = x \wedge isValidCap(last(A^s_{ed}))$$

$$Q_{11} \stackrel{\text{def}}{=} \#A^s_{ra} - 1 = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x \; \neg treeLocked \wedge \neg isRaRes \wedge \neg isEdRes \wedge isEdConfRes \wedge \neg isErConfRes \wedge \neg isIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge last(A^s_{ed}) = x \wedge last(B^s_{ed}) = y \wedge isValidCap(last(B^s_{ed}))$$

$$Q_{12} \stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x$$
$$treeLocked \wedge \neg isRaRes \wedge \neg isEdRes \wedge \neg isEdConfRes \wedge$$
$$isErConfRes \wedge negisIdfRes \wedge \neg isIrConfRes \wedge last(D) = x$$

$$Q_{13} \stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} - 1 = \#B^s_{er} = 0 \wedge last(D) = x$$
$$treeLocked \wedge \neg isRaRes \wedge \neg isEdRes \wedge \neg isEdConfRes \wedge isErConfRes \wedge$$
$$\neg isIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge last(A^s_{ra}) = x$$

$$Q_{14} \stackrel{\text{def}}{=} \#A^s_{ra} - 1 = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge last(D) = x$$
$$\neg treeLocked \wedge \neg isRaRes \wedge \neg isEdRes \wedge \neg isEdConfRes \wedge$$
$$isErConfRes \wedge \neg isIdfRes \wedge \neg isIrConfRes \wedge last(D) = x \wedge$$
$$last(A^s_{er}) = x \wedge last(B^s_{er}) = y$$

$$Q_{15} \stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} = 0 \wedge \#A^s_{ed} = \#B^s_{ed} = 0 \wedge \#A^s_{er} = \#B^s_{er} = 0 \wedge \neg treeLocked$$
$$\wedge \neg isRaRes \wedge \neg isEdRes \wedge \neg isEdConfRes \wedge isErConfRes \wedge$$
$$(isIdfRes \vee isIrConfRes) \wedge last(E) = x$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption A^i_{sh}, commitment C^i_{sh}, and channels B, D, E,A^s_{ra}, B^s_{ra}, A^s_{ed}, B^s_{ed}, A^s_{er}, and B^s_{er}.

- $\models Q_{s^i_{sh}} \rightarrow C^i_{sh}$ follows from the above definitions.
- $\models Q_{s^i_{sh}} \wedge A^i_{sh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does unset boolean variables $treeLocked$, $isRaRes$, $isEdRes$, $isEdConfRes$, $isErConfRes$, $isIdfRes$, and $isIrConfRes$. Hence, this verification holds.
- $\models Q_{l_1} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_2}) \wedge C^i_{sh} \circ (f_1 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(A, v))$ for arbitrary v. In this input transition, a communication through channel D just took place. Function g assigns the last received value from channel D to x. Function f_1 checks the variable x to find the request type and assigns boolean variables $isRaRes$, $isEdRes$, $isEdConfRes$, $isErConfRes$, $isIdfRes$, and $isIrConfRes$, accordingly. Thus, this verification holds.
- $\models Q_{l_2} \wedge isRaRes \wedge A^i_{sh} \rightarrow Q_{l_3} \circ f_2$. In this internal transition, function f_2 locks the tree. Because the condition of the transition states that it is a resource-allocation response, assumption A^i_{sh} satisfies $isValidCap(last(D))$ predicate. Thus, this verification holds.
- $\models Q_{l_3} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_4}) \wedge C^i_{sh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ra}, \sigma(x)))$. In this output transition, program I_{sh} sends x through channel A^s_{ra} toward program I^s_{ra}. C^i_{sh} holds because $\sigma \models isRaRes \wedge isValidCap(x)$. Hence, this verification holds.
- $\models Q_{l_4} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_5}) \wedge C^i_{sh} \circ (f_3 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(A, v))$ for arbitrary v. In this input transition, program I_{sh} receives a message through channel B^s_{ra} which function g assigns it to variable y. Based on the assumptions, program

I_{sh} assumes the response contains a valid capability. Furthermore, function f_3 unlocks the tree. Thus, this verification holds.

- $\models Q_{l_5} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_1}) \wedge C^i_{sh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ra}, \sigma(y)))$. In this output transition, program I_{sh} sends y through channel B toward program O. C^i_{sh} holds because $\sigma \models isRaRes \wedge isValidCap(y)$. Hence, this verification holds.

- $\models Q_{l_2} \wedge isEdRes \wedge A^i_{sh} \rightarrow Q_{l_6} \circ f_2$. In this internal transition, function f_2 locks the tree. Because the condition of the transition states that it is a external-delegation response, assumption A^i_{sh} satisfies $isValidCap(last(D))$ predicate. Thus, this verification holds.

- $\models Q_{l_6} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_7}) \wedge C^i_{sh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ed}, \sigma(x)))$. In this output transition, program I_{sh} sends x through channel A^s_{er} toward program I^s_{ed}. C^i_{sh} holds because $\sigma \models isEdRes \wedge isValidCap(x)$. Hence, this verification holds.

- $\models Q_{l_7} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_8}) \wedge C^i_{sh} \circ (f_3 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(A, v))$ for arbitrary v. In this input transition, program I_{sh} receives a message through channel B^s_{ed} which function g assigns it to variable y. Based on the assumptions, program I_{sh} assumes the response contains a valid capability. Furthermore, function f_3 unlocks the tree. Thus, this verification holds.

- $\models Q_{l_8} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_1}) \wedge C^i_{sh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ed}, \sigma(y)))$. In this output transition, program I_{sh} sends y through channel B toward program O. C^i_{sh} holds because $\sigma \models isEdRes \wedge isValidCap(y)$. Hence, this verification holds.

- $\models Q_{l_2} \wedge isEdConfRes \wedge A^i_{sh} \rightarrow Q_{l_9} \circ f_2$. In this internal transition, function f_2 locks the tree. Because the condition of the transition states that it is a external-delegation-confirmation response, assumption A^i_{sh} satisfies $isValidCap(last(D))$ predicate. Thus, this verification holds.

- $\models Q_{l_9} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_{10}}) \wedge C^i_{sh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ed}, \sigma(x)))$. In this output transition, program I_{sh} sends x through channel A^s_{ed} toward program I^s_{ed}. C^i_{sh} holds because $\sigma \models isEdConfRes \wedge isValidCap(x)$. Hence, this verification holds.

- $\models Q_{l_{10}} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_{11}}) \wedge C^i_{sh} \circ (f_3 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(A, v))$ for arbitrary v. In this input transition, program I_{sh} receives a message through channel B^s_{ed} which function g assigns it to variable y. Based on the assumptions, program I_{sh} assumes the response contains a valid capability. Furthermore, function f_3 unlocks the tree. Thus, this verification holds.

- $\models Q_{l_{11}} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_1}) \wedge C^i_{sh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ed}, \sigma(y)))$. In this output transition, program I_{sh} sends y through channel B toward program O. C^i_{sh} holds because $\sigma \models isEdConfRes \wedge isValidCap(y)$. Hence, this verification holds.

- $\models Q_{l_2} \wedge isErConfRes \wedge A^i_{sh} \rightarrow Q_{l_{12}} \circ f_2$. In this internal transition, function f_2 locks the tree. Because the condition of the transition states that it is a external-delegation-confirmation response, assumption A^i_{sh} satisfies $isValidCap(last(D))$ predicate. Thus, this verification holds.

- $\models Q_{l_{12}} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_{13}}) \wedge C^i_{sh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{er}, \sigma(x)))$. In this output transition, program I_{sh} sends x through channel A^s_{er} toward program I^s_{er}. C^i_{sh} holds because $\sigma \models isEdConfRes$. Hence, this verification holds.

- $\models Q_{l_{10}} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_{11}}) \wedge C^i_{sh} \circ (f_3 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(A, v))$ for arbitrary v. In this input transition, program I_{sh} receives a message through channel B^s_{er} which function g assigns it to variable y. Based on the assumptions, program I_{sh} assumes the response is a external-delegation-confirmation response. Furthermore, function f_3 unlocks the tree. Thus, this verification holds.

- $\models Q_{l_{11}} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_1}) \wedge C^i_{sh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ed}, \sigma(y)))$. In this output transition, program I_{sh} sends y through channel B toward program O. C^i_{sh} holds because $\sigma \models isEdConfRes$). Hence, this verification holds.

- $\models Q_{l_1} \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_{13}}) \wedge C^i_{sh} \circ (f_1 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(A, v))$ for arbitrary v. In this input transition, a communication through channel D just took place and function f_1 checks the variable x to find the request type and assigns boolean variables $isIdfRes$, and $isIrConfRes$, accordingly. Assumption A^i_{sh} states t hat the input is a internal-delegation or internal-revocation-confirmation response. Thus, this verification holds.

- $\models Q_{l_{13}} \wedge (isIdRes \vee isIrConfRes) \wedge A^i_{sh} \rightarrow ((A^i_{sh} \rightarrow Q_{l_1}) \wedge C^i_{sh}) \circ (g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(x)))$. In this output transition, program I_{sh} sends x through channel B toward program O. C^i_{sh} holds because the condition of the transition states that the message is $isIdRe$ or $isIrConfRes$. Hence, this verification holds.

Since $Q_{t^i_{sh}} \rightarrow \psi^i_{sh}$, the post-condition of program I_{sh} is satisfied.

Now, we check the validity of the hypothesis in Rule 1. Program I_{qh} receives requests from channel A and processes them in a FIFO order. Because I_{qh} assumes each request contains a unique process ID and commits to forwarding the request to the right program, it satisfies the assumption of programs I^q_{ra} (A^q_{ra}), I_{id} (A_{id}), I^q_{ed} (A^q_{ed}), I_{ir} (A_{ir}), and I^q_{er} (A^q_{er}). I_{sh} processes received responses from channel D. Because each response contains a valid capability and I_{sh} commits forwarding it to the right program. Furthermore, I_{sh} receives valid responses from programs I^s_{ra}, I^s_{ed}, and I^s_{er}, and forward them via channel B.

6.3 Parallel Composition

The instantiated access-control system (denoted by program S) comprises programs R, I, and O, running in parallel ($S \stackrel{\text{def}}{=} R \parallel I \parallel O$). Hence, We apply the parallel

composition rule to acquire the instantiated system's A-C formula as the following:

$$\vdash \langle A^s,\ C^s \rangle :\ \{\varphi^s\}\ S\ \{\psi^s\}$$

Program O commits to program I that each process uses its PID when sending a request, and the PID is unique. Furthermore, program O commits to forwarding responses to their corresponding processes directly. In addition, the commitments of programs R and I comprise their modules' commitments. Therefore, the above A-C formula proves Lemmas 1 to 5 for the instantiated system.

6.4 Proof

In this section, we proof the above lemmas for the MDC design. We proof the lemmas using the rely-guarantee technique too. In addition, we use the proofs of those lemmas as the building blocks in proofs of the lemmas for the MDC design.

To proof the lemmas, we model an operating system, an intermediate controller, and a resource controller by programs $O \stackrel{\text{def}}{=} (L_o, T_o, s_o, t_o)$, $I \stackrel{\text{def}}{=} (L_i, T_i, s_i, t_i)$, and $R \stackrel{\text{def}}{=} (L_r, T_r, s_r, t_r)$, respectively. Program I is defined as parallel execution of the programs of the modules in the intermediate controller as following:

$$I \stackrel{\text{def}}{=} I_{H_q} \parallel I_{H_s} \parallel I_{ra} \parallel I_{ld} \parallel I_{rd} \parallel I^d_{rd} \parallel I_{rr}$$

Program R is defined as parallel execution of the programs of the modules in the resource controller as following:

$$R \stackrel{\text{def}}{=} R_{h_{qs}} \parallel R_{ra} \parallel R_{rd} \parallel R_{rr}$$

Now, we proof the lemmas regarding these programs.

6.4.1 First Lemma: Resource Allocation

First lemma indicates that any resource-allocation operation creates a valid p-cap. We formally define the lemma and proof it.

Lemma 6. *Given a process with unique ID in a process node, a intermediate controller in the same node, and a resource controller in a resource node, the process will get a valid p-cap by sending a resource-allocation request.*

Proof. We use Assumption-Commitment technique to prove the lemma.

Program I_{qh} receives a request from process P_{ra} via channel A and finds out the request is a resource-allocation one. Then, I_{qh} forwards the request to program I^q_{ra}, receives the response from it, and forwards the response toward program R through channel C.

The A-C formula for program I_{qh} is as follows:

$$\vdash \langle Ass_{qh},\ C_{qh} \rangle\ \{\varphi_{qh}\}\ I_{qh}\ \{\psi_{qh}\}$$

where φ_{qh}, ψ_{qh} , Ass_{qh}, and C_{qh} are formalised as follows:

$$
\begin{array}{lll}
\varphi_{qh} & \stackrel{\text{def}}{=} & \#A = \#C = 0 \ \wedge \ \#A^q_{ra} = \#B^q_{ra} = 0 \\
\psi_{qh} & \stackrel{\text{def}}{=} & false \\
Ass_{qh} & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow isUniquePID(last(A).spid)) \wedge \\
 & & (\#B^q_{ra} > 0 \rightarrow \\
 & & \quad isRaReq(last(B^q_{ra})) \ \wedge \ isUniquePID(last(B^q_{ra}).spid)) \\
C_{qh} & \stackrel{\text{def}}{=} & (\#B > 0 \rightarrow isUniquePID(last(B).spid)) \wedge \\
 & & (\#A^q_{ra} > 0 \rightarrow \\
 & & \quad isRaReq(last(A^q_{ra})) \ \wedge \ isUniquePID(last(A^q_{ra}).spid))
\end{array}
$$

wherein N_a, N_b, and 0 are logical variables.

The assertion network for I_{qh} is as follows:

$$
\begin{array}{l}
Q_s \stackrel{\text{def}}{=} \#A = \#C = 0 \ \wedge \ \#A^q_{ra} = \#B^q_{ra} = 0 \\
Q_{l_1} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \\
Q_{l_2} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ last(A) = x \\
Q_{l_3} \stackrel{\text{def}}{=} (\#A^q_{ra} - 1) = \#B^q_{ra} \ \wedge \ last(A) = x \wedge \\
\qquad last(A^q_{ra}) = x \\
Q_{l_4} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ last(A) = x \wedge \\
\qquad last(A^q_{ra}) = x \ \wedge \ last(B^q_{ra}) = y \\
Q_t \stackrel{\text{def}}{=} false
\end{array}
$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{qh}, commitment C_{qh}, and channels A, B, A^q_{ra}, and B^q_{ra}.

- $\models Q_s \rightarrow C_{qh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{qh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \ \wedge \ Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_2}) \ \wedge \ C_{qh}) \circ (f_1 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, \ h \mapsto v, \ \sigma(h).(A, v))$ for arbitrary value $v \in MSG$. In this input transition, $\sigma' \models \#A > 0$ holds because just a communication through channel A took place. Function g assigns the received value to variable x. Since $\#A > 0$, the first implication of Ass_{qh} satisfies $isUniquePID(last(A).spid)$. Thus, this verification holds.
- $\models Q_{l_2} \ \wedge \ isRAReq(x) \ \wedge \ Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_3}) \ \wedge \ C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ra}, \sigma(x)))$. In this output transition, the request in variable x is forwarded

toward resource-allocation module through channel A^q_{ra} because $isRaReq(x)$ is evaluated to *true*. C_{qh} is satisfied because $\#A^q_{ra} > 0$, from the condition of the transition we have $isRaReq(last(A^q_{ra}))$, and from Q_{l_2} we have $isUniquePID(last(A^q_{ra}).spid)$. Thus, this verification holds.

- $\models Q_{l_3} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_4}) \wedge C_{qh}) \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y,\ h \mapsto v,\ \sigma(h).(B^q_{ra}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives the response of program I_{ra} through channel B^q_{ra} and assign it to variable y via function g. Since $\#B^q_{ra} > 0$, the first implication of Ass_{qh} implies $isRaReq(last(B^q_{ra}))$ and $isUniquePID(last(B^q_{ra}).spid)$. Thus, this verification holds.

  $\models Q_{l_4} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_1}) \wedge C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y)))$. In this output transition, the program sends the request in variable y through channel B. C_{qh} holds because from Q_{l_3} we have $isRaReq(y)$ and $isUniquePID(y.spid)$. Hence, this verification holds.

Finally, the post-condition of program I_{qh} is satisfied because $Q_t \rightarrow \psi_{qh}$.

Program R_h receives a request from program I via channel A and finds out the request is resource-allocation one. Then, R_h forwards the request to program R_{ra}, receives the response from it, and forwards the response toward program I through channel B.

The A-C formula for process R_h is as follows:

$$\vdash \langle Ass_h,\ C_h \rangle\ \{\varphi_h\}\ I_h\ \{\psi_h\}$$

where φ_h, ψ_h , Ass_h, and C_h are formalised as follows:

$$\begin{array}{lll}
\varphi_h & \stackrel{\text{def}}{=} & \#A \geq 0 \ \wedge\ \#B \geq 0 \ \wedge\ \#A_{ra} = \#B_{ra} \\
\psi_h & \stackrel{\text{def}}{=} & false \\
Ass_h & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow isUniquePID(last(A).spid)) \wedge \\
 & & (\#B_{ra} > 0 \rightarrow \\
 & & \quad isRaRes(last(B_{ra})) \ \wedge\ isValidCap(last(B_{ra}).cap) \\
C_h & \stackrel{\text{def}}{=} & (\#B > 0 \rightarrow \\
 & & \quad isRaRes(last(B)) \ \wedge\ isValidCap(last(B).cap)) \wedge \\
 & & (\#A_{ra} > 0 \rightarrow \\
 & & \quad isRaReq(last(A_{ra})) \ \wedge\ isUniquePID(last(A_{ra}).spid))
\end{array}$$

wherein 0 is a logical variable.

The assertion network for R_h is as follows:

$$\begin{array}{l}
Q_s \stackrel{\text{def}}{=} \#A = \#C = 0 \ \wedge\ \#A_{ra} = \#B_{ra} = 0 \\
Q_{l_1} \stackrel{\text{def}}{=} \#A_{ra} = \#B_{ra} \\
Q_{l_2} \stackrel{\text{def}}{=} \#A_{ra} = \#B_{ra} \ \wedge\ last(A) = x \\
Q_{l_3} \stackrel{\text{def}}{=} (\#A_{ra} - 1) = \#B_{ra} \ \wedge\ last(A) = x \ \wedge\ last(A_{ra}) = x
\end{array}$$

$$Q_{l_4} \stackrel{\text{def}}{=} \#A^q_{ra} = \#B^q_{ra} \ \wedge \ last(A) = x \ \wedge \ last(A_{ra}) = x \wedge$$
$$last(B_{ra}) = y$$
$$Q_t \stackrel{\text{def}}{=} false$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_h, commitment C_h, and channels A, B, A_{ra}, and B_{ra}.

- $\models Q_s \rightarrow C_h$ follows from the above definition.
- $\models Q_s \ \wedge \ Ass_h \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \wedge Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_2}) \wedge C_h) \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(M, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives a message through channel M and function g stores it in variable x. Based on Ass_h, we have $isUniquePID(last(x).spid))$. Thus, this verification holds.
- $\models Q_{l_2} \ \wedge \ isRAReq(x) \ \wedge \ Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_3}) \ \wedge \ C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{ra}, \sigma(x)))$. In this output transition, the request in variable x is forwarded toward resource-allocation module through channel $\#A_{ra}$ because $isRAReq(x)$ is evaluated to *true*. C_h is satisfied because $\#A_{ra} > 0$, from the condition of the transition we have $isRaReq(last(A_{ra}))$, and from Q_{l_2} we have $isUniquePID(last(A_{ra}).spid)$. Thus, this verification holds.

  $\models Q_{l_3} \wedge Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_4}) \wedge C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B_{ra}\sigma(y)))$. In this input transition, the program receives the resource-allocation response in variable y through channel B_{ra}. Hence, this verification holds.

  $\models Q_{l_4} \ \wedge \ Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_1}) \ \wedge \ C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y)))$. In this output transition, the program sends the response in variable y through channel B. C_h holds because from Ass_h we have $isRaRes(y)$ and $isUniquePID(y.spid)$. Hence, this verification holds.

Finally, the post-condition of program R_h is satisfied because $Q_t \rightarrow \psi_h$.

Program I_{sh} receives a response from program R via channel D and finds out the response is a resource-allocation response. Then, I_{sh} forwards the response to I^s_{ra} module, receives the response from the module, and forwards the response toward program O via channel B.

The A-C formula for program I_{sh} is as follows:

$$\vdash \langle Ass_{sh}, \ C_{sh} \rangle \ \{\varphi_{sh}\} \ I_{sh} \ \{\psi_{sh}\}$$

where φ_{sh}, ψ_{sh} , Ass_{sh}, and C_{sh} are formalised as follows:

$$\varphi_{sh} \stackrel{\text{def}}{=} \#C = 0 \ \wedge \ \#D = N_d \ \wedge \ \#A^s_{ra} = \#B^s_{ra} = 0 \ \wedge \ \neg hasTreeToken$$
$$\psi_{sh} \stackrel{\text{def}}{=} false$$

$$
\begin{aligned}
Ass_{sh} \quad &\stackrel{\text{def}}{=} \quad (\#C > 0 \ \wedge \ isRaRes(last(C)) \rightarrow isValidCap(last(C).cap)) \wedge \\
&\qquad (\#B^s_{ra} > 0 \rightarrow \\
&\qquad\quad isRaRes(last(B^s_{ra})) \ \wedge \ isValidCap(last(B^s_{ra}).cap)) \\
C_{sh} \quad &\stackrel{\text{def}}{=} \quad (\#D > N_d \rightarrow \\
&\qquad\quad isRaRes(last(D)) \ \wedge \ isValidCap(last(D).cap)) \wedge \\
&\qquad (\#A^s_{ra} > 0 \rightarrow \\
&\qquad\quad isRaRes(last(A^s_{ra})) \ \wedge \ isValidCap(last(A^s_{ra}).cap))
\end{aligned}
$$

wherein 0, N_d, and 0 are logical variables.

The assertion network for I_{sh} is as follows:

$$
\begin{aligned}
Q_s &\stackrel{\text{def}}{=} \#B = \#D = 0 \ \wedge \ \#A^s_{ra} = \#B^s_{ra} = 0 \ \wedge \ \neg hasTreeToken \\
Q_{l_1} &\stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} \ \wedge \ \neg hasTreeToken \\
Q_{l_2} &\stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} \ \wedge \ last(C) = x \ \wedge \ \neg hasTreeToken \\
Q_{l_3} &\stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} \ \wedge \ last(C) = x \ \wedge \ hasTreeToken \\
&\qquad isRaRes(x) \ \wedge \ isValidCap(x.cap) \\
Q_{l_4} &\stackrel{\text{def}}{=} (\#A^s_{ra} - 1) = \#B^s_{ra} \ \wedge \ last(C) = x \wedge \\
&\qquad last(A^s_{ra}) = x \ \wedge \ hasTreeToken \\
Q_{l_5} &\stackrel{\text{def}}{=} \#A^s_{ra} = \#B^s_{ra} \ \wedge \ last(C) = x \ \wedge \ last(A^s_{ra}) = x \wedge \\
&\qquad last(B^s_{ra}) = y \ \wedge \ hasTreeToken \ \wedge \ isRaRes(y) \ \wedge \ isValidCap(y.cap) \\
Q_t &\stackrel{\text{def}}{=} false
\end{aligned}
$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{sh}, commitment C_{sh}, and channels B, D, A^s_{ra}, and B^s_{ra}.

- $\models Q_s \rightarrow C_{sh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{sh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_2}) \wedge C_{sh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(M, v))$ for arbitrary value $v \in MSG$. In this input transition, just a communication through channel C took place. Function g assigns the received value to variable x, hence $last(C) = x$. Thus, this verification holds.
- $\models Q_{l_2} \wedge isRaRes(x) \wedge Ass_{sh} \rightarrow Q_{l_3} \circ f_1$. In this internal transition, function f_1 gets the tree token. Since $\#C > 0$ and $isRaRes(x)$, from Ass_{sh} we have $isValidCap(x.cap)$). Thus, this verification holds.
- $\models Q_{l_3} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_4}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^s_{ra}, \sigma(x)))$. In this output transition, the response in variable x is forwarded toward resource-allocation module through channel A^s_{ra}. C_{qh} is satisfied because $\#A^s_{ra} > 0$ and from

Q_{l_3} we have $isRaRes(last(A^s_{ra}))$ and $isValidCap(last(A^s_{ra}).cap)$. Thus, this verification holds.

- $\models Q_{l_4} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_5}) \wedge C_{sh}) \circ g$)where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B^s_{ra}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives the response of program I_{ra} through channel B^s_{ra} and assign it to variable y via function g. Since $\#B^s_{ra} > 0$, the first implication of Ass_{sh} implies $isRaRes(last(B^s_{ra}))$ and $isValidCap(last(B^s_{ra}).cap)$. Thus, this verification holds.

  $\models Q_{l_5} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_1}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(D, \sigma(y)))$. In this output transition, the program sends the response in variable y through channel D. C_{sh} holds because from Q_{l_5} we have $isRaRes(y)$ and $isValidCap(y.cap)$. In addition, function f_2 release the tree token. Hence, this verification holds.

Finally, the post-condition of program I_{sh} is satisfied because $Q_t \rightarrow \psi_{sh}$.

When program O receives the response, the program processes and forwards the response as explained in section 6.4.1. Finally, program P_{ra} adds the capability to its capability set as described in section 5.4.

□

6.4.2 Second Lemma: Internal Delegation

Second lemmas indicates that the internal delegation of a valid p-cap by a process creates a valid p-cap for another process inside the same node. Hence, the intermediate controller handles the internal delegation inside the node. We formally define the lemma and proof it.

Lemma 7. *Given two processes with unique IDs inside a process node and a intermediate controller in the same node, if the first process delegates a valid p-cap to second one, then the second process will get a valid p-cap as the result of the delegation request.*

Proof. We use Assumption-Commitment technique to prove the lemma.

The proof of program O is explained in section 6.4.1.

Program I_{qh} receives a request from the operating system via channel A and finds out the request is internal delegation one. Then, I_{qh} forwards the request to program I_{id}, receives the response from it, and forwards the response toward the response handler through channel E.

The A-C formula for program I_{qh} is as follows:

$$\vdash \langle Ass_{qh},\ C_{qh} \rangle\ \{\varphi_{qh}\}\ I_{qh}\ \{\psi_{qh}\}$$

The assertion network for I_{qh} is as follows:

$$Q_s \stackrel{\text{def}}{=} \#A = 0 \ \wedge \ \#E = 0 \ \wedge \ \#A_{id} = \#B_{id} = \#C_{id} = 0 \wedge \neg hasTreeToken$$

$$Q_{l_1} \stackrel{\text{def}}{=} \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ \neg hasTreeToken$$

$$Q_{l_2} \stackrel{\text{def}}{=} \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ last(A) = x$$

$$Q_{l_5} \stackrel{\text{def}}{=} \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ last(A) = x \wedge isIdRes(last(A)) \ \wedge \ hasTreeToken$$

$$Q_{l_6} \stackrel{\text{def}}{=} (\#A_{id} - 1) = \#B_{id} = \#C_{id} \ \wedge \ last(A) = x \wedge last(A_{id}) = x$$

$$Q_{l_7} \stackrel{\text{def}}{=} \#A_{id} = \#B_{id} = (\#C_{id} + 1) \ \wedge \ last(A) = x \wedge last(A_{id}) = x \ \wedge \ last(B_{id}) = y$$

$$Q_{l_8} \stackrel{\text{def}}{=} \#A_{id} = \#B_{id} = (\#C_{id} + 1) \ \wedge \ last(A) = x \wedge last(A_{id}) = x \ \wedge \ last(B_{id}) = y \ \wedge \ last(E) = y$$

$$Q_{l_9} \stackrel{\text{def}}{=} \#A_{id} = \#B_{id} = \#C_{id} \ \wedge \ last(A) = x \wedge last(A_{id}) = x \ \wedge \ last(B_{id}) = y \ \wedge \ last(C_{id}) = z \ \wedge \ last(E) = y$$

$$Q_t \stackrel{\text{def}}{=} false$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{qh}, commitment C_{qh}, and channels A, E, A_{id}, B_{id}, and C_{ra}.

- $\models Q_s \rightarrow C_{qh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{qh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \ \wedge \ Ass_{qh} \ \rightarrow \ ((Ass_{qh} \ \rightarrow \ Q_{l_2}) \ \wedge \ C_{qh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, \ h \mapsto v, \ \sigma(h).(A, v))$ for arbitrary value $v \in MSG$. In this input transition, $\sigma' \models \#A > 0$ holds because just a communication through channel A took place. Function g assigns the received value to variable x. Since $\#A > 0$, the first implication of Ass_{qh} satisfies $isUniquePID(last(A).spid)$. Thus, this verification holds.
- $\models Q_{l_2} \wedge isIdRes(x) \wedge Ass_{sh} \rightarrow Q_{l_5} \circ f_1$. In this internal transition, function f_1 gets the tree token. Since $\#A > 0$, the condition of the transition we have $isIdRes(last(A))$. Thus, this verification holds.
- $\models Q_{l_5} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_6}) \wedge C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{id}, \sigma(x)))$. In this output transition, the request in variable x is forwarded toward internal-delegation module through channel A_{id} because from the condition we have $isIdRes(x)$. C_{qh} is satisfied because $\#A_{id} > 0$ and from Q_{l_5} we have $isIdRes(last(A_{id}))$. Hence,

the sender and the receiver processes have unique IDs and are in the current node. Thus, this verification holds.

- $\models Q_{l_6} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_7}) \wedge C_{qh}) \circ g)$where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B_{id}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives the response of program I_{id} through channel B_{id} and assigns it to variable y via function g. Since $\#B_{id} > 0$, the second implication of Ass_{qh} implies $isIdRes(last(B_{id}))$ and $isValidCap(last(B_{cap}).cap)$. Thus, this verification holds.

  $\models Q_{l_7} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_8}) \wedge C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(E, \sigma(y)))$. In this output transition, the program sends the response in variable y through channel E. In addition, C_{qh} holds because from Q_{l_7} and Ass_{qh} we have $isIdRes(y)$ and $isValidCap(last(y).cap)$. Hence, this verification holds.

- $\models Q_{l_8} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_8}) \wedge C_{qh}) \circ g)$where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z, h \mapsto v, \sigma(h).(C_{id}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives another response from program I_{id} through channel C_{id} and assigns it to variable z via function g. Since $\#C_{id} > 0$, the third implication of Ass_{qh} implies $isIdConfRes(last(C_{id}))$ and $isValidCap(last(C_{cap}).cap)$. Thus, this verification holds.

  $\models Q_{l_9} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_1}) \wedge C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(E, \sigma(z)))$. In this output transition, the program sends the response in variable z through channel E. C_{qh} holds because from Ass_{qh} we have $isIdConfRes(z)$ and $isValidCap(z.cap)$. In addition, function f_2 release the tree token. Hence, this verification holds.

Finally, the post-condition of program I_{qh} is satisfied because $Q_t \rightarrow \psi_{qh}$.

Program I_{sh} receives two responses from program I_{qh} via channel E. In this case, I_{sh} just forwards the responses to program O via channel D.

The A-C formula for program I_{sh} is as follows:

$$\vdash \langle Ass_{sh},\ C_{sh} \rangle\ \{\varphi_{sh}\}\ I_{sh}\ \{\psi_{sh}\}$$

where φ_{qh}, ψ_{qh} , Ass_{qh}, and C_{qh} are formalised as follows:

$$\varphi_{sh} \stackrel{\text{def}}{=} \#E = \#D = 0$$
$$\psi_{sh} \stackrel{\text{def}}{=} false$$
$$Ass_{sh} \stackrel{\text{def}}{=} true$$
$$C_{sh} \stackrel{\text{def}}{=} true$$

The assertion network for I_{sh} is as follows:

$$Q_s \stackrel{\text{def}}{=} \#E = \#D = 0$$
$$Q_{l_1} \stackrel{\text{def}}{=} \#E \geq 0 \ \wedge\ \#D \geq 0$$
$$Q_{l_{15}} \stackrel{\text{def}}{=} \#E > 0 \ \wedge\ \#D \geq 0 \ \wedge\ last(E) = x$$

$$Q_t \stackrel{\text{def}}{=} false$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{sh}, commitment C_{sh}, and channels E and D.

- $\models Q_s \rightarrow C_{sh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{sh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_15}) \wedge C_{sh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(E, v))$ for arbitrary value $v \in MSG$. In this input transition, just a communication through channel E took place. Function g assigns the received value to variable x, hence $last(E) = x$. Thus, this verification holds.
- $\models Q_{l_15} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_1}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(D, \sigma(x)))$. In this output transition, the response in variable x is just forwarded toward program O through channel D. Thus, this verification holds.

Finally, the post-condition of program I_{H_s} is satisfied because $Q_t \rightarrow \psi_{h_s}$.

When program O receives both responses, the program processes and forwards the responses as explained in section 6.4.1. Finally, programs P_1 and P_2 add the confirmation and the capability to their capability sets as described in section 5.5.

□

6.4.3 Third Lemma: External Delegation

Third lemmas indicates that the external delegation of a valid p-cap by a process creates a valid p-cap for another process inside another node. Hence, both intermediate controllers of the process nodes and the resource controller of the resource node handle the external delegation together. We formally define the lemma and proof it.

Lemma 8. *Given two processes with unique IDs in different process nodes, delegation of a valid capability from the first process to the second process causes that the delegating and the receiving processes receive a valid indicator and a valid p-cap, respectively.*

Proof. We use Assumption-Commitment technique to prove the lemma.

Program I_{qh} receives a request from the delegator program via channel A and finds out the request is a external-delegation one. Then, I_{qh} forwards the request to program I^q_{ed}, receives the response from it, and forwards the response toward program R through channel C.

The A-C formula for program I_{qh} is as follows:

$$\vdash \langle Ass_{qh},\ C_{qh} \rangle\ \{\varphi_{qh}\}\ I_{qh}\ \{\psi_{qh}\}$$

where φ_{qh}, ψ_{qh} , Ass_{qh}, and C_{qh} are formalised as follows:

$$
\begin{array}{lll}
\varphi_{qh} & \stackrel{\text{def}}{=} & \#A = \#C = 0 \ \wedge \ \#A^q_{ed} = \#B^q_{ed} \\
\psi_{qh} & \stackrel{\text{def}}{=} & false \\
Ass_{qh} & \stackrel{\text{def}}{=} & (\#A > 0 \ \wedge \ isEdReq(last(A)) \rightarrow \\
 & & \quad isUniquePID(last(A).spid) \ \wedge \ isValidCap(last(A).cap)) \wedge \\
 & & (\#B^q_{ed} > 0 \rightarrow \\
 & & \quad isEdReq(last(B^q_{ed})) \ \wedge \ isValidCap(last(B^q_{ed}).cap)) \\
C_{qh} & \stackrel{\text{def}}{=} & (\#B > 0 \rightarrow isEdReq(last(B)) \ \wedge \ isValidCap(last(B).cap)) \wedge \\
 & & (\#A^q_{ed} > 0 \rightarrow \\
 & & \quad isEdReq(last(A^q_{ed})) \ \wedge \ isValidCap(last(A^q_{ed}).cap))
\end{array}
$$

The assertion network for I_{qh} is as follows:

$$
\begin{array}{l}
Q_s \stackrel{\text{def}}{=} \#A = \#C = 0 \ \wedge \ \#A^q_{ed} = \#B^q_{ed} = N_{ed} \\
Q_{l_1} \stackrel{\text{def}}{=} \#A^q_{ed} = \#B^q_{ed} \\
Q_{l_2} \stackrel{\text{def}}{=} \#A^q_{ed} = \#B^q_{ed} \ \wedge \ last(A) = x \\
Q_{l_3} \stackrel{\text{def}}{=} (\#A^q_{ed} - 1) = \#B^q_{ed} \ \wedge \ last(A) = x \wedge \\
\qquad last(A^q_{ed}) = x \\
Q_{l_4} \stackrel{\text{def}}{=} \#A^q_{ed} = \#B^q_{ed} \ \wedge \ last(A) = x \wedge \\
\qquad last(A^q_{ed}) = x \ \wedge \ last(B^q_{ed}) = y \\
Q_t \stackrel{\text{def}}{=} false
\end{array}
$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{qh}, commitment C_{qh}, and channels A, B, A_{ra_1}, and B_{ra_1}.

- $\models Q_s \rightarrow C_{qh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{qh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \ \wedge \ Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_2}) \ \wedge \ C_{qh}) \circ (f_1 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, \ h \mapsto v, \ \sigma(h).(A, v))$ for arbitrary value $v \in MSG$. In this input transition, $\sigma' \models \#A > 0$ holds because just a communication through channel A took place. Function g assigns the received value to variable x. Since $\#A > 0$, the first implication of Ass_{qh} satisfies $isUniquePID(last(A).spid)$ and $isValidCap(last(A).cap)$. Thus, this verification holds.
- $\models Q_{l_2} \ \wedge \ isEdReq(x) \ \wedge \ Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_{10}}) \ \wedge \ C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^q_{ed}, \sigma(x)))$. In this output transition, the request in variable x is forwarded

toward external-delegation module through channel A^q_{ed} because $isEdReq(x)$ is evaluated to *true*. C_{qh} is satisfied because $\#A^q_{ed} > 0$, from the condition of the transition we have $isEdReq(last(A^q_{ed}))$, and from Q_{l_2} we have $isUniquePID(last(A^q_{ed}).spid)$. Thus, this verification holds.

- $\models Q_{l_{10}} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_{11}}) \wedge C_{qh}) \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y,\ h \mapsto v,\ \sigma(h).(B^q_{ed}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives the response of program I_{ed} through channel B^q_{ed} and assign it to variable y via function g. Since $\#B^q_{ed} > 0$, the first implication of Ass_{qh} implies $isEdReq(last(B^q_{ed}))$ and $isValidCap(last(B^q_{ed}).cap)$. Thus, this verification holds.

  $\models Q_{l_{11}} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_1}) \wedge C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y)))$. In this output transition, the program sends the request in variable y through channel B. C_{qh} holds because from $Q_{l_{11}}$ we have $isEdReq(y)$ and $isValidCap(y.cap)$. Hence, this verification holds.

Finally, the post-condition of program I_{qh} is satisfied because $Q_t \rightarrow \psi_{qh}$.

Program R_h receives a request from program I via channel A and finds out the request is external-delegation one. Then, R_h forwards the request to program R_{ed}, receives the response from it, and forwards the response toward program I through channel B.

The A-C formula for process R_h is as follows:

$$\vdash \langle Ass_h,\ C_h \rangle\ \{\varphi_h\}\ I_h\ \{\psi_h\}$$

where φ_h, ψ_h , Ass_h, and C_h are formalised as follows:

$$\begin{aligned}
\varphi_h &\stackrel{\text{def}}{=} \#A\#B \geq 0 \ \wedge\ \#A_{ed} = \#B_{bd} = \#C_{ed} \\
\psi_h &\stackrel{\text{def}}{=} false \\
Ass_h &\stackrel{\text{def}}{=} (\#A > 0 \ \wedge\ isEdReq(last(A)) \rightarrow \\
&\quad isUniquePID(last(A).spid) \ \wedge\ isValidCap(last(A).cap)) \wedge \\
&\quad (\#B_{ra} > N_{ra} \rightarrow \\
&\quad isEdRes(last(B_{ed})) \ \wedge\ isValidCap(last(B_{ed}).cap) \\
&\quad (\#C_{ed} > 0 \rightarrow \\
&\quad isEdConfReq(last(C_{ed})) \ \wedge\ isValidCap(last(C_{ed}).cap)) \\
C_h &\stackrel{\text{def}}{=} (\#B > 0 \rightarrow \\
&\quad isEdReq(last(B)) \ \wedge\ isValidCap(last(B).cap)) \wedge \\
&\quad (\#A_{ra} > N_{ra} \rightarrow \\
&\quad isEdReq(last(A_{ra})) \ \wedge\ isUniquePID(last(A_{ra}).spid))
\end{aligned}$$

The assertion network for R_h is as follows:

$$\begin{aligned}
Q_s &\stackrel{\text{def}}{=} \#A = \#B = 0 \ \wedge\ \#A_{ed} = \#B_{ed} = \#C_{ed} \\
Q_{l_1} &\stackrel{\text{def}}{=} \#A_{ra} = \#B_{ra} = \#C_{ed}
\end{aligned}$$

$$Q_{l_2} \stackrel{\text{def}}{=} \#A_{ed} = \#B_{ed} = \#C_{ed} \ \wedge \ last(A) = x$$

$$Q_{l_5} \stackrel{\text{def}}{=} (\#A_{ra} - 1) = \#B_{ra} = \#C_{ed} \ \wedge \ last(A) = x \ \wedge \ last(A_{ed}) = x$$

$$Q_{l_6} \stackrel{\text{def}}{=} \#A^q_{ed} = \#B^q_{ed} = (\#C_{ed} + 1) \ \wedge \ last(A) = x \ \wedge \ last(A_{ed}) = x \ \wedge \ last(B_{ed}) = y$$

$$Q_{l_7} \stackrel{\text{def}}{=} \#A^q_{ed} = \#B^q_{ed} \ \wedge \ last(A) = x \ \wedge \ last(A_{ed}) = x \ \wedge \ last(B_{ed}) = y \ \wedge \ last(B) = y$$

$$Q_{l_8} \stackrel{\text{def}}{=} \#A^q_{ed} = \#B^q_{ed} \ \wedge \ last(A) = x \ \wedge \ last(A_{ed}) = x \ \wedge \ last(B_{ed}) = y \ \wedge \ last(C_{ed}) = z \ \wedge \ last(B) = y$$

$$Q_t \stackrel{\text{def}}{=} false$$

- $\models Q_s \rightarrow C_h$ follows from the above definition.
- $\models Q_s \ \wedge \ Ass_h \rightarrow Q_{l_1} \ \circ \ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \wedge Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_2}) \wedge C_h) \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(M, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives a message through channel M and function g stores it in variable x. Based on Ass_h, we have $isUniquePID(last(x).spid))$. Thus, this verification holds.
- $\models Q_{l_2} \ \wedge \ isEdReq(x) \ \wedge \ Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_5}) \ \wedge \ C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{ed}, \sigma(x)))$. In this output transition, the request in variable x is forwarded toward resource-allocation module through channel $\#A_{ed}$ because $isEdReq(x)$ is evaluated to *true*. C_h is satisfied because $\#A_{ed} > 0$, from the condition of the transition we have $isEdReq(last(A_{ed}))$, and from Q_{l_2} we have $isUniquePID(last(A_{ed}).spid)$. Thus, this verification holds.

  $\models Q_{l_5} \ \wedge \ Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_6}) \ \wedge \ C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B_{ed}\sigma(y)))$. In this input transition, the program receives the delegation response in variable y through channel B_{ed}. Hence, this verification holds.

  $\models Q_{l_6} \wedge Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_7}) \wedge C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y)))$. In this output transition, the program sends the response in variable y through channel B. C_h holds because from Q_{l_6} and Ass_h we have $isEdRes(y)$ and $isValidCap(y.cap)$. Hence, this verification holds.

  $\models Q_{l_7} \ \wedge \ Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_8}) \ \wedge \ C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C_{ed}, \sigma(z)))$. In this input transition, the program receives the delegation-confirmation response in variable z through channel C_{ed}. Hence, this verification holds.

  $\models Q_{l_8} \wedge Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_1}) \wedge C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(z)))$. In this output transition, the program sends the response in variable z through channel B. C_h holds because from Q_{l_8} and Ass_h we have $isEdConfRes(z)$ and $isValidCap(y.cap)$. Hence, this verification holds.

Finally, the post-condition of program R_h is satisfied because $Q_t \rightarrow \psi_h$.

Program I_{sh} on the receiving-process side receives a response from program R via channel C and finds out it is a delegation response. Then, I_{sh} forwards the response to RD_3 module, receives the response from the module, and forwards the response toward program O via channel D.

The A-C formula for program I_{sh} is as follows:

$$\vdash \langle Ass_{sh},\ C_{sh}\rangle\ \{\varphi_{sh}\}\ I_{sh}\ \{\psi_{sh}\}$$

where φ_{qh}, ψ_{qh} , Ass_{qh}, and C_{qh} are formalised as follows:

$$\begin{array}{lll}
\varphi_{sh} & \stackrel{\text{def}}{=} & \#C = \#D = 0\ \wedge\ \#A_{rd_3} = \#B_{rd_3}\ \wedge\ \neg hasTreeToken \\
\psi_{sh} & \stackrel{\text{def}}{=} & false \\
Ass_{sh} & \stackrel{\text{def}}{=} & (\#C > 0\ \wedge\ isEdRes(last(C)) \rightarrow isValidCap(last(C).cap)) \wedge \\
 & & (\#B_{rd_3} > 0 \rightarrow \\
 & & \quad isEdRes(last(B_{rd_3}))\ \wedge\ isValidCap(last(B_{rd_3}).cap)) \\
C_{sh} & \stackrel{\text{def}}{=} & (\#D > 0\ \wedge\ isEdRes(last(D)) \rightarrow isValidCap(last(D).cap)) \wedge \\
 & & (\#A_{rd_3} > 0 \rightarrow \\
 & & \quad isEdRes(last(A_{rd_3}))\ \wedge\ isValidCap(last(A_{rd_3}).cap))
\end{array}$$

The assertion network for I_{sh} is as follows:

$$\begin{array}{l}
Q_s \stackrel{\text{def}}{=} \#C \geq 0\ \wedge\ \#D \geq 0\ \wedge\ \#A_{rd_3} = \#B_{rd_3}\ \wedge\ \neg hasTreeToken \\
Q_{l_1} \stackrel{\text{def}}{=} \#C \geq 0\ \wedge\ \#D \geq 0\ \wedge\ \#A_{rd_3} = \#B_{rd_3}\ \wedge\ \neg hasTreeToken \\
Q_{l_2} \stackrel{\text{def}}{=} \#C > 0\ \wedge\ \#D \geq 0\ \wedge\ \#A_{rd_3} = \#B_{rd_3}\ \wedge\ last(C) = x\ \wedge\ \neg hasTreeToken \\
Q_{l_6} \stackrel{\text{def}}{=} \#C > 0\ \wedge\ \#D \geq 0\ \wedge\ \#A_{rd_3} = \#B_{rd_3}\ \wedge\ last(C) = x\ \wedge\ hasTreeToken \\
\qquad isEdRes(x)\ \wedge\ isValidCap(x.cap) \\
Q_{l_7} \stackrel{\text{def}}{=} \#C > 0\ \wedge\ \#D \geq 0\ \wedge\ (\#A_{rd_3} - 1) = \#B^s_{ed}\ \wedge\ last(C) = x\ \wedge \\
\qquad last(A_{rd_3}) = x\ \wedge\ hasTreeToken \\
Q_{l_8} \stackrel{\text{def}}{=} \#C > 0\ \wedge\ \#D \geq 0\ \wedge\ \#A_{rd_3} = \#B_{rd_3}\ \wedge\ last(C) = x\ \wedge\ last(A_{rd_3}) = x\ \wedge \\
\qquad last(B_{rd_3}) = y\ \wedge\ hasTreeToken\ \wedge\ isEdRes(y)\ \wedge\ isValidCap(y.cap) \\
Q_t \stackrel{\text{def}}{=} false
\end{array}$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{sh}, commitment C_{sh}, and channels C, D, A_{rd_3}, and B_{rd_3}.

- $\models Q_s \rightarrow C_{sh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{sh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.

- $\models Q_{l_1} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_2}) \wedge C_{sh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(C, v))$ for arbitrary value $v \in MSG$. In this input transition, just a communication through channel C took place. Function g assigns the received value to variable x, hence $last(C) = x$. Thus, this verification holds.

- $\models Q_{l_2} \wedge isEdRes(x) \wedge Ass_{sh} \rightarrow Q_{l_6} \circ f_1$. In this internal transition, function f_1 gets the tree token. Since $\#C > 0$ and $isEdRes(x)$, from Ass_{sh} we have $isValidCap(x.cap))$. Thus, this verification holds.

- $\models Q_{l_6} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_7}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{rd_3}, \sigma(x)))$. In this output transition, the response in variable x is forwarded toward resource-allocation module through channel A_{rd_3}. C_{qh} is satisfied because $\#A_{rd_3} > 0$ and from Q_{l_6} we have $isEdRes(last(A_{rd_3}))$ and $isValidCap(last(A_{rd_3}).cap)$. Thus, this verification holds.

- $\models Q_{l_7} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_9}) \wedge C_{sh}) \circ g)$where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B_{rd_3}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives the response of program RD_3 through channel B_{rd_3} and assign it to variable y via function g. Since $\#B_{rd_3} > 0$, the first implication of Ass_{sh} implies $isEdRes(last(B^s_{ed}))$ and $isValidCap(last(B_{rd_3}).cap)$. Thus, this verification holds.

  $\models Q_{l_9} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_1}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(D, \sigma(y)))$. In this output transition, the program sends the response in variable y through channel D. C_{sh} holds because from Ass_{sh} we have $isRaRes(B_{rd_3})$ and $isValidCap(last(B_{rd_3}).cap)$. In addition, function f_2 release the tree token. Hence, this verification holds.

Finally, the post-condition of program I_{sh} is satisfied because $Q_t \rightarrow \psi_{sh}$.

When program O receives both responses, the program processes and forwards the responses as explained in section 6.4.1. Then, programs P' receives and adds the new capability to its capability set as described in section 5.6.

Program I_{sh} on the delegating-process side receives a response from program R via channel C and finds out it is a delegation-confirmation response. Then, I_{sh} forwards the response to RD_2 module, receives the response from the module, and forwards the response toward program O via channel D.

The A-C formula for program I_{sh} is as follows:

$$\vdash \langle Ass_{sh},\ C_{sh} \rangle\ \{\varphi_{sh}\}\ I_{sh}\ \{\psi_{sh}\}$$

$$\begin{aligned}
\varphi_{sh} &\stackrel{\text{def}}{=} \#C > 0 \ \wedge\ \#D > 0 \ \wedge\ \#A^s_{ed} = \#B^s_{ed} \ \wedge\ \neg hasTreeToken \\
\psi_{sh} &\stackrel{\text{def}}{=} false \\
Ass_{sh} &\stackrel{\text{def}}{=} (\#C > 0 \ \wedge\ isEdConfRes(last(C)) \rightarrow isValidCap(last(C).cap)) \wedge \\
&\qquad (\#B^s_{ed} > 0 \rightarrow \\
&\qquad\quad isEdConfRes(last(B^s_{ed})) \ \wedge\ isValidCap(last(B^s_{ed}).cap)) \\
C_{sh} &\stackrel{\text{def}}{=} (\#D > 0 \ \wedge\ isEdConfRes(last(D)) \rightarrow isValidCap(last(D).cap)) \wedge
\end{aligned}$$

$$(\#A^s_{ed} > 0 \rightarrow \\ isEdConfRes(last(A^s_{ed})) \ \wedge \ isValidCap(last(A^s_{ed}).cap))$$

The assertion network for I_{sh} is as follows:

$$Q_s \stackrel{\text{def}}{=} \#C \geq 0 \ \wedge \ \#D \geq 0 \ \wedge \ \#A^s_{ed} = \#B^s_{ed} \ \wedge \ \neg hasTreeToken$$

$$Q_{l_1} \stackrel{\text{def}}{=} \#C \geq 0 \ \wedge \ \#D \geq 0 \ \wedge \ \#A^s_{ed} = \#B^s_{ed} \ \wedge \ \neg hasTreeToken$$

$$Q_{l_2} \stackrel{\text{def}}{=} \#C > 0 \ \wedge \ \#D \geq 0 \ \wedge \ \#A^s_{ed} = \#B^s_{ed} \ \wedge \ last(C) = x \ \wedge \ \neg hasTreeToken$$

$$Q_{l_9} \stackrel{\text{def}}{=} \#C > 0 \ \wedge \ \#D \geq 0 \ \wedge \ \#A^s_{ed} = \#B^s_{ed} \ \wedge \ last(C) = x \ \wedge \ hasTreeToken \\ isEdConfRes(x) \ \wedge \ isValidCap(x.cap)$$

$$Q_{l_{10}} \stackrel{\text{def}}{=} \#C > 0 \ \wedge \ \#D \geq 0 \ \wedge \ (\#A^s_{ed} - 1) = \#B^s_{ed} \ \wedge \ last(C) = x \wedge \\ last(A^s_{ed}) = x \ \wedge \ hasTreeToken$$

$$Q_{l_{11}} \stackrel{\text{def}}{=} \#C > 0 \ \wedge \ \#D \geq 0 \ \wedge \ \#A^s_{ed} = \#B^s_{ed} \ \wedge \ last(C) = x \ \wedge \ last(A^s_{ed}) = x \wedge \\ last(B^s_{ed}) = y \ \wedge \ hasTreeToken \ \wedge \ isEdConfRes(y) \wedge \\ isValidCap(y.cap)$$

$$Q_t \stackrel{\text{def}}{=} false$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{sh}, commitment C_{sh}, and channels C, D, A^s_{ed}, and B^s_{ed}.

- $\models Q_s \rightarrow C_{sh}$ follows from the above definition.
- $\models Q_s \ \wedge \ Ass_{sh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_2}) \wedge C_{sh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(C, v))$ for arbitrary value $v \in MSG$. In this input transition, just a communication through channel C took place. Function g assigns the received value to variable x, hence $last(C) = x$. Thus, this verification holds.
- $\models Q_{l_2} \ \wedge \ isEdConfRes(x) \ \wedge \ Ass_{sh} \rightarrow Q_{l_6} \circ f_1$. In this internal transition, function f_1 gets the tree token. Since $\#C > 0$ and $isEdRes(x)$, from Ass_{sh} we have $isValidCap(x.cap)$). Thus, this verification holds.
- $\models Q_{l_6} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_7}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^s_{ed}, \sigma(x)))$. In this output transition, the response in variable x is forwarded toward resource-allocation module through channel A^s_{ed}. C_{qh} is satisfied because $\#A^s_{ed} > 0$ and from Q_{l_6} we have $isEdConfRes(last(A^s_{ed}))$ and $isValidCap(last(A^s_{ed}).cap)$. Thus, this verification holds.
- $\models Q_{l_7} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_9}) \wedge C_{sh}) \circ g)$where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B^s_{ed}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives the

response of program RD_2 through channel B^s_{ed} and assign it to variable y via function g. Since $\#B^s_{ed} > 0$, the first implication of Ass_{sh} implies $isEdConfRes(last(B^s_{ed}))$ and $isValidCap(last(B^s_{ed}).cap)$. Thus, this verification holds.

$\models Q_{l_9} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_1}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(D, \sigma(y)))$. In this output transition, the program sends the response in variable y through channel D. C_{sh} holds because from Ass_{sh} we have $isEdConfRes(B^s_{ed})$ and $isValidCap(last(B^s_{ed}).cap)$. In addition, function f_2 release the tree token. Hence, this verification holds.

Finally, the post-condition of program I_{sh} is satisfied because $Q_t \rightarrow \psi_{sh}$.

When program O receives the response, the program processes and forwards the response as explained in section 6.4.1. Finally, program P adds the confirmation capability to its capability set as described in section 5.6.

□

6.4.4 Forth Lemma: Internal Revocation

Forth lemmas indicates that the internal revocation of a valid p-cap by the delegator process will invalidate the delegated p-cap to another process inside the same node. Since the p-cap is delegated internally, the intermediate controller handles the revocation alone. We formally define the lemma and proof it.

Lemma 9. *Given a process with unique ID in a process node and a intermediate controller in the same process node, revoking a delegated p-cap to another process inside the same node invalidates the delegated p-cap .*

Proof. We use Assumption-Commitment technique to prove the lemma.

Program I_{qh} receives the request of a process via channel A and finds out the request is internal revocation one. Then, I_{qh} forwards the request to I_{ir} module, receives the response from it, and forwards the response toward the response handler through channel E. The A-C formula for program I_{qh} is as follows:

$$\vdash \langle Ass_{qh},\ C_{qh} \rangle\ \{\varphi_{qh}\}\ I_{qh}\ \{\psi_{qh}\}$$

$$\begin{aligned}
\varphi_{qh} &\stackrel{\text{def}}{=} \#A = \#E = 0 \ \wedge\ \#A_{ir} = \#B_{ir} = N_{ir} \wedge \\
&\quad \neg hasTreeToken \\
\psi_{qh} &\stackrel{\text{def}}{=} false \\
Ass_{qh} &\stackrel{\text{def}}{=} (\#A > 0 \ \wedge\ isIrReq(Last(A)) \rightarrow \\
&\quad isUniquePID(last(A).spid)) \ \wedge\ isValidCap(last(A).cap)) \wedge \\
&\quad (\#B_{ir} > 0 \rightarrow isIrConfRes(last(B_{ir})) \\
C_{qh} &\stackrel{\text{def}}{=} (\#E > 0 \rightarrow isIrConfRes(last(E)) \wedge \\
&\quad (\#A_{ir} > 0 \rightarrow isIrReq(last(A_{ir})) \ \wedge\ isValidCap(last(A_{ir}).cap)
\end{aligned}$$

wherein N_{ir} is a logical variable.
The assertion network for I_{qh} is as follows:

$$Q_s \stackrel{\text{def}}{=} \#A = \#E \geq 0 \ \wedge \ \#A_{ir} = \#B_{ir} = 0 \wedge \neg hasTreeToken$$

$$Q_{l_1} \stackrel{\text{def}}{=} \#A_{ir} = \#B_{ir} \ \wedge \ \neg hasTreeToken$$

$$Q_{l_2} \stackrel{\text{def}}{=} \#A_{ir} = \#B_{ir} \ \wedge \ last(A) = x$$

$$Q_{l_{12}} \stackrel{\text{def}}{=} \#A_{ir} = \#B_{ir} = \#C_{ir} \ \wedge \ last(A) = x \wedge isIrReq(last(A)) \ \wedge \ hasTreeToken$$

$$Q_{l_{13}} \stackrel{\text{def}}{=} (\#A_{ir} - 1) = \#B_{ir} \ \wedge \ last(A) = x \ \wedge \ last(A_{ir}) = x$$

$$Q_{l_{14}} \stackrel{\text{def}}{=} \#A_{ir} = \#B_{ir} \ \wedge \ last(A) = x \ \wedge \ last(A_{ir}) = x \wedge last(B_{ir}) = y$$

$$Q_t \stackrel{\text{def}}{=} false$$

- $\models Q_s \rightarrow C_{qh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{qh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_2}) \wedge C_{qh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(A, v))$ for arbitrary value $v \in MSG$. In this input transition, $\sigma' \models \#A > 0$ holds because just a communication through channel A took place. Function g assigns the received value to variable x. Thus, this verification holds.
- $\models Q_{l_2} \ \wedge \ isLDReq(x) \ \wedge \ Ass_{sh} \rightarrow Q_{l_{12}} \ \circ \ f_1$. In this internal transition, function f_1 gets the tree token. In addition, we have $isUniquePID(last(A).spid))$ and $isValidCap(last(A).cap))$ because $\#A > 0$ and the condition states that $isIrReq(last(A))$. Thus, this verification holds.
- $\models Q_{l_{12}} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_{13}}) \wedge C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{ir}, \sigma(x)))$. In this output transition, the request in variable x is forwarded toward internal-revocation module through channel A_{ir} because from the condition we have $isIrReq(x)$. C_{qh} is satisfied because $\#A_{ir} > 0$, $last(A) = last(A_{ir}) = x$, and from Ass_{qh} we have $isUniquePID(last(A).spid))$ and $isValidCap(last(A).cap))$. Thus, this verification holds.
- $\models Q_{l_{13}} \ \wedge \ Ass_{qh} \ \rightarrow \ ((Ass_{qh} \ \rightarrow \ Q_{l_{14}}) \ \wedge \ C_{qh}) \circ g)$where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, \ h \mapsto v, \ \sigma(h).(B_{ir}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives the response of program I_{ir} through channel B_{ir} and assigns it to variable y via function g. Since $\#B_{ir} > 0$, the second implication of Ass_{qh} implies $isIrConfRes(last(B_{ir}))$. Thus, this verification holds.

  $\models Q_{l_{14}} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_1}) \wedge C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(E, \sigma(y)))$. In this output transition, the program sends the response in variable y through channel

E. C_{qh} holds because from Ass_{qh} we have $isIrConfRes(y)$. In addition, function f_2 release the tree token. Hence, this verification holds.

Finally, the post-condition of program I_{qh} is satisfied because $Q_t \rightarrow \psi_{qh}$.

Program I_{sh} receives the response from program I_{qh} via channel E. In this case, I_{sh} just forwards the response to program O via channel B.

The A-C formula for program I_{sh} is as follows:

$$\vdash \langle Ass_{sh},\ C_{sh}\rangle\ \{\varphi_{sh}\}\ I_{sh}\ \{\psi_{sh}\}$$

where φ_{sh}, ψ_{sh} , Ass_{sh}, and C_{sh} are formalised as follows:

$$\varphi_{sh} \stackrel{\text{def}}{=} \#E = \#B = 0$$
$$\psi_{sh} \stackrel{\text{def}}{=} false$$
$$Ass_{sh} \stackrel{\text{def}}{=} true$$
$$C_{sh} \stackrel{\text{def}}{=} true$$

The assertion network for I_{sh} is as follows:

$$Q_s \stackrel{\text{def}}{=} \#E = \#B = 0$$
$$Q_{l_1} \stackrel{\text{def}}{=} \#E \geq 0 \ \wedge\ \#B \geq 0$$
$$Q_{l_15} \stackrel{\text{def}}{=} \#E > 0 \ \wedge\ \#B \geq 0 \ \wedge\ last(E) = x$$
$$Q_t \stackrel{\text{def}}{=} false$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{sh}, commitment C_{sh}, and channels E and B.

- $\models Q_s \rightarrow C_{sh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{sh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_15}) \wedge C_{sh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(E, v))$ for arbitrary value $v \in MSG$. In this input transition, just a communication through channel E took place. Function g assigns the received value to variable x, hence $last(E) = x$. Thus, this verification holds.
- $\models Q_{l_15} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_1}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(D, \sigma(x)))$. In this output transition, the response in variable x is just forwarded toward program O through channel D. Thus, this verification holds.

Finally, the post-condition of program I_{sh} is satisfied because $Q_t \rightarrow \psi_{sh}$.

When program O receives both responses, the program processes and forwards the responses as explained in section 6.4.1. Finally, programs P receives the confirmation and delete the capability from its capability set as described in section **??**.

□

6.4.5 Fifth Lemma: External Revocation

Fifth lemmas indicates that the external revocation of a valid p-cap by a delegator process will invalidate the delegated p-cap to the receiving process inside another node. In this operation, the intermediate controller on the delegator-process side and the resource controller of the resource node handle the external revocation together. We formally define the lemma and proof it.

Lemma 10. *Given a process with a unique ID in a process node, revocation of a valid delegated capability by the process will revoke the delegated p-cap.*

Proof. We use Assumption-Commitment technique to prove the lemma.

Program I_{qh} receives the request from a process via channel A and finds out the request is external revocation one. Then, I_{qh} forwards the request to program R, receives the response from it, and forwards the response toward program R through channel C. The A-C formula for program I_{qh} is as follows:

$$\vdash \langle Ass_{qh},\ C_{qh} \rangle\ \{\varphi_{qh}\}\ I_{qh}\ \{\psi_{qh}\}$$

$$\begin{aligned}
\varphi_{qh} &\stackrel{\text{def}}{=} \#A = \#C = 0\ \wedge\ \#A^q_{er} = \#B^q_{er} \\
\psi_{qh} &\stackrel{\text{def}}{=} false \\
Ass_{qh} &\stackrel{\text{def}}{=} (\#A > 0\ \wedge\ isErReq(Last(A))) \rightarrow \\
&\quad isUniquePID(last(A).spid))\ \wedge\ isValidCap(last(A).cap)) \wedge \\
&\quad (\#B_{rr} > 0 \rightarrow isErReq(last(B_{ld}))) \\
C_{qh} &\stackrel{\text{def}}{=} (\#B > 0 \rightarrow isErReq(last(O))) \wedge \\
&\quad (\#A^q_{er} > 0 \rightarrow isErReq(last(A^q_{er}))\ \wedge\ isUniquePID(last(A).spid)) \wedge \\
&\quad isValidCap(last(A_{ld}).cap))
\end{aligned}$$

The assertion network for I_{qh} is as follows:

$$\begin{aligned}
Q_s &\stackrel{\text{def}}{=} \#A = \#B = 0\ \wedge\ \#A^q_{er} = \#B^q_{er} = 0 \\
Q_{l_1} &\stackrel{\text{def}}{=} \#A^q_{er} = \#B^q_{er} \\
Q_{l_2} &\stackrel{\text{def}}{=} \#A^q_{er} = \#B^q_{er}\ \wedge\ last(A) = x \\
Q_{l_{15}} &\stackrel{\text{def}}{=} (\#A^q_{er} - 1) = \#B^q_{er}\ \wedge\ last(A) = x \wedge \\
&\quad last(A^q_{er}) = x \\
Q_{l_{16}} &\stackrel{\text{def}}{=} \#A^q_{er} = \#B^q_{er}\ \wedge\ last(A) = x \wedge \\
&\quad last(A^q_{er}) = x\ \wedge\ last(B^q_{er}) = y \\
Q_t &\stackrel{\text{def}}{=} false
\end{aligned}$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{qh}, commitment C_{qh}, and channels A, C, A^q_{er}, and B^q_{er}.

- $\models Q_s \rightarrow C_{qh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{qh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_2}) \wedge C_{qh}) \circ (f_1 \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x,\ h \mapsto v,\ \sigma(h).(A, v))$ for arbitrary value $v \in MSG$. In this input transition, $\sigma' \models \#A > 0$ holds because just a communication through channel A took place. Function g assigns the received value to variable x. Thus, this verification holds.
- $\models Q_{l_2} \wedge isErReq(x) \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_{15}}) \wedge C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{rr}, \sigma(x)))$. In this output transition, the request in variable x is forwarded toward resource-allocation module through channel A_{rr} because $isErReq(x)$ is evaluated to *true*. C_{qh} is satisfied because $\#A_{rr} > 0$, from the condition of the transition we have $isErReq(last(A_{rr}))$, and from Ass_{qh} we have $isUniquePID(last(A_{rr}).spid)$ and $isValidCap(last(A_{rr}).cap)$. Thus, this verification holds.
- $\models Q_{l_{15}} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_{16}}) \wedge C_{qh}) \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y,\ h \mapsto v,\ \sigma(h).(B_{rr}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives the response of program I_{rr} through channel B_{rr} and assign it to variable y via function g. Since $\#B_{rr} > 0$, the first implication of Ass_{qh} implies $isErReq(last(B_{rr}))$ and $isValidCap(last(B_{rr}).cap)$. Thus, this verification holds.

  $\models Q_{l_{16}} \wedge Ass_{qh} \rightarrow ((Ass_{qh} \rightarrow Q_{l_1}) \wedge C_{qh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y)))$. In this output transition, the program sends the request in variable y through channel B. C_{qh} holds because we have $isErReq(y)$ and $isValidCap(y.cap)$. Hence, this verification holds.

Finally, the post-condition of program I_{H_q} is satisfied because $Q_t \rightarrow \psi_{h_q}$.

Program R_h receives a request from program I via channel A and finds out the request is external-revocation one. Then, R_h forwards the request to program R_{er}, receives the response from it, and forwards the response toward program I through channel B.

The A-C formula for process R_h is as follows:

$$\vdash \langle Ass_h,\ C_h \rangle\ \{\varphi_h\}\ I_h\ \{\psi_h\}$$

where φ_h, ψ_h , Ass_h, and C_h are formalised as follows:

$$\begin{array}{lll} \varphi_h & \stackrel{\text{def}}{=} & \#A = \#B = 0 \ \wedge\ \#A_{er} = \#B_{er} \\ \psi_h & \stackrel{\text{def}}{=} & false \\ Ass_h & \stackrel{\text{def}}{=} & (\#A > 0 \ \wedge\ isErReq(last(B_{rr})) \rightarrow isValidCap(last(A).cap))) \wedge \\ & & (\#B_{er} > 0 \rightarrow \end{array}$$

$$
\begin{aligned}
& isErRes(last(B_{er})) \ \wedge \ isErConfRes(last(B_{er}).cap) \\
C_h \quad \stackrel{\text{def}}{=} \quad & (\#B > 0 \rightarrow \\
& \quad isErConfRes(last(B))) \wedge \\
& (\#A_{er} > 0 \rightarrow \\
& \quad isErRes(last(A_{er})) \ \wedge \ isValidCap(last(A_{er}.cap).spid))
\end{aligned}
$$

The assertion network for R_h is as follows:

$$
\begin{aligned}
Q_s &\stackrel{\text{def}}{=} \#A = \#B = 0 \ \wedge \ \#A_{er} = \#B_{er} \\
Q_{l_1} &\stackrel{\text{def}}{=} \#A = \#B \ \wedge \ \#A_{er} = \#B_{er} \\
Q_{l_2} &\stackrel{\text{def}}{=} \#A_{er} = \#B_{er} \ \wedge \ last(A) = x \\
Q_{l_3} &\stackrel{\text{def}}{=} (\#A_{er} - 1) = \#B_{er} \ \wedge \ last(A) = x \ \wedge \ last(A_{er}) = x \\
Q_{l_4} &\stackrel{\text{def}}{=} \#A_{er} = \#B_{er} \ \wedge \ last(A) = x \ \wedge \ last(A_{er}) = x \wedge \\
&\qquad last(B_{er}) = y \\
Q_t &\stackrel{\text{def}}{=} false
\end{aligned}
$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_h, commitment C_h, and channels A, B, A_{er}, and B_{er}.

- $\models Q_s \rightarrow C_h$ follows from the above definition.
- $\models Q_s \ \wedge \ Ass_h \rightarrow Q_{l_1} \ \circ \ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.
- $\models Q_{l_1} \wedge Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_2}) \wedge C_h) \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(M, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives a message through channel A and function g stores it in variable x. Based on Ass_h, we have $isUniquePID(last(x).spid))$. Thus, this verification holds.
- $\models Q_{l_2} \ \wedge \ isRAReq(x) \ \wedge \ Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_3}) \ \wedge \ C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A_{er}, \sigma(x)))$. In this output transition, the request in variable x is forwarded toward resource-allocation module through channel $\#A_{er}$ because $isRAReq(x)$ is evaluated to *true*. C_h is satisfied because $\#A_{er} > 0$, from the condition of the transition we have $isRaReq(last(A_{er}))$, and from Q_{l_2} we have $isUniquePID(last(A_{er}).spid)$. Thus, this verification holds.

  $\models Q_{l_3} \wedge Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_4}) \wedge C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B_{er}\sigma(y)))$. In this input transition, the program receives the resource-allocation response in variable y through channel B_{er}. Hence, this verification holds.

  $\models Q_{l_4} \wedge Ass_h \rightarrow ((Ass_h \rightarrow Q_{l_1}) \wedge C_h))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y)))$. In this output transition, the program sends the response in variable y through channel

B. C_h holds because from Ass_h we have $isErConfRes(y)$. Hence, this verification holds.

Finally, the post-condition of program R_h is satisfied because $Q_t \rightarrow \psi_h$.

Program I_{sh} on the process side receives a response from program R via channel D and finds out it is a revocation-confirmation response. Then, I_{sh} forwards the response to program I^s_{er} module, receives the response from the module, and forwards the response toward program O via channel B.

The A-C formula for program I_{sh} is as follows:

$$\vdash \langle Ass_{sh},\ C_{sh}\rangle\ \{\varphi_{sh}\}\ I_{sh}\ \{\psi_{sh}\}$$

$$\begin{aligned}
\varphi_{sh} &\stackrel{\text{def}}{=} \#D = \#B = 0\ \wedge\ \#A^s_{er} = \#B^s_{er}\ \wedge\ \neg hasTreeToken\\
\psi_{sh} &\stackrel{\text{def}}{=} false\\
Ass_{sh} &\stackrel{\text{def}}{=} (\#D > 0\ \wedge\ isErConfRes(last(C)) \rightarrow isValidCap(last(C).cap)) \wedge\\
&\quad\ (\#B^s_{er} > 0 \rightarrow isErConfRes(last(B^s_{er})))\\
C_{sh} &\stackrel{\text{def}}{=} (\#B > 0 \rightarrow isErConfRes(last(D))) \wedge\\
&\quad\ (\#A^s_{er} > 0 \rightarrow isErConfRes(last(A^s_{er})))
\end{aligned}$$

The assertion network for I_{sh} is as follows:

$$\begin{aligned}
Q_s &\stackrel{\text{def}}{=} \#D = \#B = 0\ \wedge\ \#A^s_{er} = \#B^s_{er}\ \wedge\ \neg hasTreeToken\\
Q_{l_1} &\stackrel{\text{def}}{=} \#A^s_{er} = \#B^s_{er}\ \wedge\ \neg hasTreeToken\\
Q_{l_2} &\stackrel{\text{def}}{=} \#A^s_{er} = \#B^s_{er}\ \wedge\ last(C) = x\ \wedge\ \neg hasTreeToken\\
Q_{l_{12}} &\stackrel{\text{def}}{=} \#A^s_{er} = \#B^s_{er}\ \wedge\ last(C) = x\ \wedge\ hasTreeToken\\
&\quad\ isErConfRes(x)\\
Q_{l_{13}} &\stackrel{\text{def}}{=} (\#A^s_{er} - 1) = \#B^s_{er}\ \wedge\ last(C) = x \wedge\\
&\quad\ last(A^s_{er}) = x\ \wedge\ hasTreeToken\\
Q_{l_{14}} &\stackrel{\text{def}}{=} \#A^s_{er} = \#B_{rd_2}\ \wedge\ last(C) = x\ \wedge\ last(A^s_{er}) = x \wedge\\
&\quad\ last(B^s_{er}) = y\ \wedge\ hasTreeToken\ \wedge\ isErConfRes(y)\\
Q_t &\stackrel{\text{def}}{=} false
\end{aligned}$$

We check the following verifications to prove that above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass_{sh}, commitment C_{sh}, and channels B, D, A^s_{er}, and B^s_{er}.

- $\models Q_s \rightarrow C_{sh}$ follows from the above definition.
- $\models Q_s \wedge Ass_{sh} \rightarrow Q_{l_1} \circ init$. In this internal transition, function *init* does not change the state of the program. Hence, this verification holds.

- $\models Q_{l_1} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_2}) \wedge C_{sh}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x, h \mapsto v, \sigma(h).(C, v))$ for arbitrary value $v \in MSG$. In this input transition, just a communication through channel C took place. Function g assigns the received value to variable x, hence $last(C) = x$. Thus, this verification holds.

- $\models Q_{l_2} \wedge isRDConfRes(x) \wedge Ass_{sh} \rightarrow Q_{l_{12}} \circ f_1$. In this internal transition, function f_1 gets the tree token. Since $\#C > 0$ from Ass_{sh} we have $isErConfRes(x)$. Thus, this verification holds.

- $\models Q_{l_{12}} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_{13}}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(A^s_{er}, \sigma(x)))$. In this output transition, the response in variable x is forwarded toward external-revocation module through channel A^s_{er}. C_{qh} is satisfied because $\#A^s_{er} > 0$ and from Ass_{sh} we have $isErConfRes(last(A^s_{er}))$. Thus, this verification holds.

- $\models Q_{l_{13}} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_{14}}) \wedge C_{sh}) \circ g)$where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : y, h \mapsto v, \sigma(h).(B^s_{er}, v))$ for arbitrary value $v \in MSG$. In this input transition, the program receives the response of program RR_2 through channel B^s_{er} and assign it to variable y via function g. Since $\#B^s_{er} > 0$, the first implication of Ass_{sh} implies $isErConfRes(last(B^s_{er}))$. Thus, this verification holds.

  $\models Q_{l_{14}} \wedge Ass_{sh} \rightarrow ((Ass_{sh} \rightarrow Q_{l_1}) \wedge C_{sh}))$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(D, \sigma(y)))$. In this output transition, the program sends the response in variable y through channel D. C_{sh} holds because from Ass_{sh} we have $isErConfRes(B^s_{er})$. In addition, function f_2 release the tree token. Hence, this verification holds.

Finally, the post-condition of program I_{sh} is satisfied because $Q_t \rightarrow \psi_{sh}$.

When program O receives the response, the program processes and forwards the response as explained in section 6.4.1. Finally, program P adds the confirmation capability to its capability set as described in section 5.8.

□

Table 3: Correctness Checks of the Models

Model	Transitions	Depth	Time(sec)	Lines of Code
RA	2224	127	0.01	350
ID	950	106	0.01	349
ED	496828	277	2.15	700
IR	1165	86	0.01	287
ER	26403	140	0.06	380

7 Model Checking

We use the SPIN model checker [27] to check our design. SPIN employs the *PROMELA* language to define models. We built the models of Lemmas 1 to 5 using PROMELA, where each model consists of the following PROMELA object types.

Processes: a process is a *proctype* object type that defines the behavior of programs in the lemmas, such as Ira and Rra (Figure 6).

Channels: a channel models a synchronous messaging channel in the A-C method, using *chan* object type. a channel is a one-way semantic interface between two programs in the lemmas (Figure 6).

Data Objects: PROMELA only supports basic object types, such as *byte* and *short*, but not complex ones, such as *floating-point*.

Claims: a system verification proves what is possible in a model and what is not. PROMELA defines a model's logical correctness using five claim types: *basic assertions*, *meta labels*, *never claims*, and *trace assertions*. We employ basic assertions and trace assertions to check the correctness of our models according to the assumption-commitment method. A Basic assertion includes an expression that PSIN evaluates as False or True during a verification; a False assertion causes the verification to fail. A trace assertion checks if the model has exchanged messages in a specific order and if the exchanged messages include defined data.

We modeled and verified Lemmas 1 to 5, for the general access control system and the MDC instantiated system; SPIN verified all the models as correct. Table 3 depicts the details of each verification

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A new kernel foundation for UNIX development. (1986).

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ouster-

hout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[4] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. *In FAST* (2014).

[5] Leonid Azriel, Lukas Humbel, Reto Achermann, Alex Richardson, Moritz Hoffmann, Avi Mendelson, Timothy Roscoe, Robert NM Watson, Paolo Faraboschi, and Dejan Milojicic. 2019. Memory-side protection with a capability enforcement co-processor. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (2019), 1–26.

[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 29–44.

[7] Daniel S Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D Hill, et al. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* 43, 2 (2023), 30–38.

[8] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. 2015. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 1–9.

[9] Kirk M Bresniker, Paolo Faraboschi, Avi Mendelson, Dejan Milojicic, Timothy Roscoe, and Robert NM Watson. 2019. Rack-scale capabilities: fine-grained protection for large-scale memories. *Computer* 52, 2 (2019), 52–62.

[10] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. 2020. Fluidmem: Full, flexible, and fast memory disaggregation for the cloud. In *IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 665–677.

[11] Nicholas P Carter, Stephen W Keckler, and William J Dally. 1994. Hardware support for fast capability-based addressing. *ACM SIGOPS Operating Systems Review* 28, 5 (1994), 319–327.

[12] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, et al. 2022. Enzian: an open, general, CPU/FPGA platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 434–451.

[13] George W Cox, William M Corwin, Konrad K Lai, and Fred J Pollack. 1981. A unified model and implementation for interprocess communication in a multiprocessor environment. In *Proceedings of the eighth ACM symposium on Operating systems principles*. 125–126.

[14] Compute Express Link (CXL). 2023 (accessed April 16, 2023). CXL Specification. `https://www.computeexpresslink.org/download-the-specification`.

[15] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. 2001. Grid information services for distributed resource sharing. In *Proceedings 10th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 181–194.

[16] W-P De Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. 2001. *Concurrency verification: Introduction to compositional and non-compositional methods*. Vol. 54. Cambridge University Press.

[17] Jack B Dennis and Earl C Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155.

[18] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 103–114.

[19] Alan Ehret, Jacob Abraham, Mihailo Isakov, and Michel A Kinsy. 2022. Zeno: A scalable capability-based secure architecture. *arXiv preprint arXiv:2208.09800* (2022).

[20] Robert S. Fabry. 1974. Capability-based addressing. *Commun. ACM* 17, 7 (1974), 403–412.

[21] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 249–264.

[22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[24] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 417–433.

[25] Norman Hardy. 1985. KeyKOS architecture. *ACM SIGOPS Operating Systems Review* 19, 4 (1985), 8–25.

[26] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. 2019. Semperos: A distributed capability system. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 709–722.

[27] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.

[28] Merle E Houdek, Frank G Soltis, and Roy L Hoffman. 1981. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th annual symposium on Computer Architecture*. 341–348.

[29] Anita K Jones, Robert J Chansler Jr, Ivor Durham, Karsten Schwans, and Steven R Vegdahl. 1979. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the seventh ACM symposium on Operating systems principles*. 117–127.

[30] Kostas Katrinis, Dimitris Syrivelis, Dionisios Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, Kobi Hasharoni, Daniel Raho, Christian Pinto, F Espina, et al. 2016. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 690–695.

[31] Kimberly Keeton. 2015. The machine: An architecture for memory-centric computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, Vol. 10.

[32] Kimberly Keeton. 2017. Memory-Driven Computing.. In *FAST*.

[33] Vamsee Reddy Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. 2021. Deact: Architecture-aware virtual memory support for fabric attached memory systems. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 453–466.

[34] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 488–504.

[35] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanaël Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, et al. 2017. Understanding {Rack-Scale} Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*.

[36] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al.

2023. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[37] Jochen Liedtke. 1995. On micro-kernel construction. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 237–250.

[38] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news* 37, 3 (2009), 267–278.

[39] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12.

[40] Rui Lin, Yuxin Cheng, Marilet De Andrade, Lena Wosinska, and Jiajia Chen. 2020. Disaggregated data centers: Challenges and trade-offs. *IEEE Communications Magazine* 58, 2 (2020), 20–26.

[41] Sergio Maffeis, John C Mitchell, and Ankur Taly. 2010. Object capabilities and isolation of untrusted web applications. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 125–140.

[42] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation*.

[43] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. 2003. *Capability myths demolished*. Technical Report. Technical Report SRL2003-02, Johns Hopkins University Systems Research ....

[44] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 106–122.

[45] Jayadev Misra and K. Mani Chandy. 1981. Proofs of networks of processes. *IEEE transactions on software engineering* 4 (1981), 417–426.

[46] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. 1990. Amoeba: A distributed operating system for the 1990s. *Computer* 23, 5 (1990), 44–53.

[47] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage.. In *NSDI, Volume 17*. 17–33.

[48] Roger M Needham and Robin DH Walker. 1977. The Cambridge CAP computer and its protection system. *ACM SIGOPS Operating Systems Review* 11, 5 (1977), 1–10.

[49] Richard Otter. 1948. The number of trees. *Annals of Mathematics* (1948), 583–599.

[50] Antonios D Papaioannou, Reza Nejabati, and Dimitra Simeonidou. 2016. The benefits of a disaggregated data centre: A resource allocation approach. In *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–7.

[51] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. 2020. Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 868–880.

[52] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.

[53] Vineet Rajani, Deepak Garg, and Tamara Rezk. 2016. On access control, capabilities, their equivalence, and confused deputy attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, 150–163.

[54] Richard F Rashid and George G Robertson. 1981. Accent: A communication oriented network operating system kernel. *ACM SIGOPS Operating Systems Review* 15, 5 (1981), 64–75.

[55] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. ReDMArk: Bypassing RDMA Security Mechanisms.. In *USENIX Security Symposium*. 4277–4292.

[56] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, et al. 1992. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*. Seattle WA (USA), 39–70.

[57] Jerome H Saltzer. 1974. Protection and the control of information sharing in Multics. *Commun. ACM* 17, 7 (1974), 388–402.

[58] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. 1999. EROS: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*. 170–185.

[59] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks.. In *NSDI*. 255–270.

[60] Richard J Swan, Samuel H Fuller, and Daniel P Siewiorek. 1977. Cm* a modular, multi-microprocessor. In *Proceedings of the June 13-16, 1977, national computer conference*. 637–644.

[61] Konstantin Taranov, Benjamin Rothenberger, Daniele De Sensi, Adrian Perrig, and Torsten Hoefler. 2022. NeVerMore: Exploiting RDMA Mistakes in NVMe-oF Storage Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2765–2778.

[62] Shin-Yeh Tsai and Yiying Zhang. 2019. A {Double-Edged} Sword: Security Threats and Opportunities in {One-Sided} Network Communication. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*. 3–3.

[63] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. 2022. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. *arXiv preprint arXiv:2211.02682* (2022).

[64] Shicheng Wang, Menghao Zhang, Yuying Du, Ziteng Chen, Zhiliang Wang, Mingwei Xu, Renjie Xie, and Jiahai Yang. 2024. Lordma: A new low-rate dos attack in rdma networks. *in NDSS* (2024).

[65] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 457–468.

[66] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson, and Fred Pollack. 1974. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM* 17, 6 (1974), 337–345.