UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

MASTER THESIS

MASTER IN SOFTWARE AND SYSTEMS

CONCURRENT LIBRARY ABSTRACTION WITHOUT INFORMATION HIDING

AUTHOR: ARTEM KHYZHA
SUPERVISOR: MANUEL CARRO LIÑARES
CO-SUPERVISOR: ALEXEY GOTSMAN

MADRID — JULY, 2014

Abstract

The commonly accepted approach to specifying libraries of concurrent algorithms is a library abstraction. Its idea is to relate a library to another one that abstracts away from details of its implementation and is simpler to reason about. A library abstraction relation has to validate the Abstraction Theorem: while proving a property of the client of the concurrent library, the library can be soundly replaced with its abstract implementation. Typically a library abstraction relation, such as linearizability, assumes a complete information hiding between a library and its client, which disallows them to communicate by means of shared memory. However, such way of communication may be used in a program, and correctness of interactions on a shared memory depends on the implicit contract between the library and the client.

In this work we approach library abstraction without any assumptions about information hiding. To be able to formulate the contract between components of the program, we augment machine states of the program with two abstract states, *views*, of the client and the library. It enables formalising the contract with the *internal safety*, which requires components to preserve each other's views whenever their command is executed. We define the library abstraction relation by establishing a correspondence between possible uses of a concrete and an abstract library. For our library abstraction relation and traces of a program, components of which follow their contract, we prove an Abstraction Theorem.

Resumen

La técnica más aceptada actualmente para la especificación de librerías de algoritmos concurrentes es la abstracción de librerías (library abstraction). La idea subyacente es relacionar la librería original con otra que abstrae los detalles de implementación y con la que es más fácil razonar formalmente. Una relación que describa dicha abstracción de librerías debe validar el Teorema de Abstracción: durante la prueba de la validez de una propiedad del cliente de la librería concurrente, el reemplazo de esta última por su implementación abstracta es lógicamente correcto. Usualmente, una relación de abstracción de librerías como la linearizabilidad (linearizability), tiene como premisa el ocultamiento de información entre el cliente y la librería (information hiding), es decir, que no se les permite comunicarse mediante la memoria compartida. Sin embargo, dicha comunicación ocurre en la práctica y la correctitud de estas interacciones en una memoria compartida depende de un contrato implícito entre la librería y el cliente.

En este trabajo, se propone un nueva definición del concepto de abtracción de librerías que no presupone un ocultamiento de información entre la librería y el cliente. Con el fin de establecer un contrato entre diferentes componentes de un programa, extendemos la máquina de estados subyacente con dos estados abstractos que representan las vistas del cliente y la librería. Esto permite la formalización de la propiedad de seguridad interna (internal safety), que requiere que cada componente preserva la vista del otro durante la ejecuci on de un comando. Consecuentemente, se define la relación de abstracción de librerías mediante una correspondencia entre los usos posibles de una librería abstracta y una concreta. Finalmente, se prueba el Teorema de Abstracción para la relación de abstracción de librerías propuesta, para cualquier traza de un programa y cualquier componente que satisface los contratos apropiados.

Contents

Chapter 1

Introduction

The modern software normally is developed modularly by encapsulating frequently used pieces of code into libraries. The typical examples of that are standard libraries of programming languages, which allow developers to benefit from using well checked implementations of algorithms and data structures such as those from java.util.concurrent for Java. The modularity becomes even more important, when the reused code is error prone and difficult to reason about, which notoriously is the case for concurrent algorithms and non-blocking data structures. To simplify reasoning about concurrent software, we need to exploit the available modularity. In particular, while reasoning about a client of a concurrent library, we would like to abstract away from the details of a particular library implementation. This requires an appropriate notion of library correctness.

Correctness of concurrent libraries is commonly formalised by linearizability [3], which fixes a certain correspondence between the library and its specification that is called *library abstraction*. The specification is usually just another library, but implemented atomically using an abstract data type. A good notion of linearizability should validate an Abstraction Theorem [4]: it is sound to replace a library with its specification in reasoning about its client.

The classical linearizability assumes a complete isolation between a library and its client, with interactions limited to passing values of a given data type as parameters or return values of library methods. However, for many applications this assumption is too restrictive, because many non-blocking algorithms use different ways of communication between components, i.e. they operate on the shared memory concurrently. In Section 2 we provide an example of such an algorithm. Overall, there is a need in defining library abstraction under weaker assumptions, when client and library communicate by means of shared memory.

In this work we approach the problem assuming that components of the program do not hide any data from each other and share the whole address space. Although the shared memory can be accessed and modified concurrently by both client and library, typically components follow a certain contract that regulates communications between them. For instance, consider a memory cell being used by a client as a counter. The contract that client may require from libraries is that no library

decreases this counter. In order to formalise the contract between the client and the library, we augment executions of the program with abstract states, *views*, which are provided by the Views Framework [2] and can be instantiated into assertions of Hoare-style logics for reasoning about concurrent programs, such as Concurrent Separation Logic [7] or Rely-Guarantee [5].

Each component has its own view at each step of the execution. The intuitive meaning of a component's view is a declaration of all possible transitions in it. When all components are considered together, their views must be compatible with a machine state and with each other, meaning that no transition of one component invalidates views of the other. In Chapter 4 we formalise this property and call it the *internal safety* of the program. Similarly, we define *internally safe semantics* of programs that consists of the traces in which contracts between components are followed.

We formulate the library abstraction relation and prove it to be sound in the Abstraction Theorem (Definition 12 and Theorem 13, Chapter 5). We also extend our result from internally safe semantics to usual traces (Corollary 20, Chapter 5) for closed programs.

Chapter 2

Motivation

In this chapter we give an example of the program, in which components communicate by means of the shared memory. We consider a well-known implementation of a non-blocking concurrent queue due to Michael and Scott [6]. The queue algorithm uses a custom memory allocator for nodes in the linked list representing the queue. To avoid the allocator becoming a performance bottleneck, Michael and Scott also implement it using a non-blocking algorithm – a concurrent stack due to Treiber [8]. In this program the non-blocking queue is the client and the allocator is the library.

The parts of the memory owned by the queue and the allocator are not disjoint for the following reason. Non-blocking algorithms often need to check that the information on the basis of which a change to a data structure was computed is still valid when the data structure is updated accordingly. This is usually done by ensuring that certain fields in the data structure have not been changed since the last time the thread making the update read them. The CAS operation allows checking that a field still has the old value atomically with the update. However, this equality does not always imply that the field has not been changed. Namely, a so-called ABA problem may arise, when the data structure is changed from its original state A to B, and then restored to A again. For example, a queue node can be returned to the allocator, and then allocated and inserted into the data structure again. To avoid this problem, the queue and the allocator use modification counters, atomically incremented at every write to certain memory cells in the data structures, which allows distinguishing between the two As in ABA. However, in the case when a validation of the update fails due to a modification counter mismatch, the read of the counter might access a node that is no longer present in the data structure, i.e., has been returned to the allocator in the case of the queue, or has been allocated to the queue in the case of the allocator. Consequently, at least modification counters must be shared by the client and the library in this example.

We futher present the example in more detail. Michael and Scott's queue is presented in Figure 2.1. For illustration, in the example we use the C-like language. A client using the implementation can call several `enqueue` or `dequeue` operations concurrently. The queue is non-blocking, i.e., implemented with compare-and-swap (CAS) operations instead of locks. CAS takes three arguments: a memory address

```
struct Node { NodeRef next; int val; };
struct NodeRef { Node *ptr; unsigned count; };
NodeRef head, tail;

void init() {
  head.ptr = (Node*)alloc();
  head.ptr->next = NULL;
  head.count = 0;
  tail = head;
}

int enqueue(int val) {
  Node *node;
  NodeRef next, last;
  node = alloc();
  if (node == NULL) return FAIL;
  node->val = val;
  node->next.ptr = NULL;
  while (true) {
    atomic { last = tail; }
    atomic { next = last.ptr->next; }
    if (atomic { tail != last }) continue;
    if (next.ptr == NULL) {
      if (CAS(last.ptr->next, next,
              NodeRef(node, next.count+1)))
        break;
    } else
      CAS(tail, last, NodeRef(next.ptr, last.count+1));
  }
  CAS(tail, last, NodeRef(next.ptr, last.count+1));
  return SUCCESS;
}

int dequeue() {
  NodeRef next, first, last;
  int val;
  while (true) {
    atomic { first = head; }
    atomic { last = tail; }
    atomic { next = first.ptr->next; }
    if (atomic { head != first }) continue;
    if (first.ptr == last.ptr) {
      if (next.ptr == NULL) return EMPTY;
      CAS(tail, last, NodeRef(next.ptr, last.count+1));
    } else {
      atomic { val = next->val; }
      if (CAS(head, first,
              NodeRef(next.ptr, first.count+1)))
        break;
    }
  }
  free(first.ptr);
  return val;
}
```

Figure 2.1: Michael and Scott's non-blocking queue [6]

```
struct Block { Block *next; };
struct BlockRef { Block *ptr; unsigned count; };
BlockRef Top;

void free(void *block) {
  BlockRef t, x;
  do {
    atomic { t = Top; }
    (Block*)block->next = t.ptr;
    x = BlockRef((Block*)block, t.count+1);
  } while (!CAS(&Top, t, x));
}

void *alloc() {
  BlockRef t, x;
  do {
    atomic { t = Top; }
    if (t.ptr == NULL)
      return NULL;
    x = BlockRef(t.ptr->next, t.count+1);
  } while (!CAS(&Top, t, x));
  return t.ptr;
}
```

Figure 2.2: A concurrent memory allocator implemented using Treiber's stack [8]

`addr`, an expected value `v1`, and a new value `v2`. It atomically reads the memory address and updates it with the new value when the address contains the expected value; otherwise, it does nothing. In C syntax `CAS(addr, v1, v2)` might be written as follows:

```
atomic {
  if (*addr==v1) { *addr=v2; return 1; }
  else { return 0; }
}
```

In most architectures an efficient CAS (or an equivalent operation) is provided natively by the processor.

Like most concurrent algorithms with explicit memory management, the queue algorithm uses a custom memory allocator to allocate `Node` structures, which Michael and Scott implement using a concurrent non-blocking stack due to Treiber [8] (Figure 2.2). We first explain this algorithm, as the simpler one of the two.

Memory allocator. The allocator stores the free list of memory blocks of size `sizeof(Node)` as a linked list, pointed to by the variable `Top`. The pointer to the next element of the list is stored at the beginning of each block. We assume `sizeof(Node)` $\geq$ `sizeof(Block*)` + `sizeof(unsigned)`, so that the pointer can be stored without overwriting the last counter field of the structure `Node`. As we noted in above, the queue algorithm relies on this field not being changed by the memory allocator even after a node is deallocated. For brevity, we omitted the initialisation code.

The operations on the list are implemented as follows. The `free` operation *(i)*

reads the current value of the top-of-the-stack pointer `Top`; *(ii)* stores the read value of `Top` at the beginning of the block `block` being deallocated; and *(iii)* atomically updates the top-of-the-stack pointer with the new value `block`. If the pointer has changed between *(i)* and *(iii)*, the CAS fails and the operation is restarted. The `alloc` operation is implemented in a similar way. Note that it returns `NULL` when the allocator is out of memory.

To avoid the *ABA* problem, the algorithm associates a counter with the `Top` variable, incremented on every modification. This ensures that, when a CAS succeeds, the `Top` variable has not changed since it was read by the thread that executed it: the counter excludes the possibility of the variable being changed temporarily and then restored to the previous value. We assume that the size of the `NodeRef` structure is of a size such that it can be read and written to atomically. Following Michael and Scott [6], we assume that the modification counter is unbounded, which is clearly idealistic. However, a version of this algorithm with a bounded counting mechanism does get used in practice, e.g., in Java memory management[1]. In this case, a bound is picked such that an overflow will (hopefully) not occur, and a bounded counter will be equivalent to an unbounded one.

Consider an execution of the `alloc` method in which it is preempted in between reading `Top` and `t.ptr->next`. Another `alloc` method invocation might run to completion, removing the memory block `t.ptr` points to and returning a pointer to it to the client of the allocator. When the first `alloc` method wakes up, it will thus read a memory cell that is being used by the client. This does not cause a problem, since the allocator only reads the cell, but not writes to it, and free cells are never returned to the operating system. However, this means that in our proof we cannot consider the state of the allocator as being completely disjoint from the state of its client, motivating the approach to library abstraction that we take in this work.

Non-blocking queue. We now give an explanation of the Michael and Scott's queue algorithm[2]. The algorithm in Figure 2.1 implements the queue as a singly-linked list with `head` and `tail` pointers. The `head` pointer always points to a dummy node, which is the first node in the list; `tail` points to either the last or second to last node in the list. The implementation calls the allocator to create new nodes in the list representing the queue. The `enqueue` operation returns `FAIL` if the allocator runs out of memory, and `SUCCESS` in all other cases. Like the allocator implementation, this algorithm uses modification counters to avoid the *ABA* problem, this time for all nodes in the queue. It relies on the fact that modification counters only increase, and are not modified by the memory allocator. As before, the queue implementation might end up reading from a memory cell freed to the allocator (but not writing to it).

The `enqueue` operation takes place in two distinct steps. Normally, the `enqueue` method creates a new node by calling the memory allocator, locates the last node in the queue, and performs the following two steps:

[1] D. F. Bacon. Parallel and concurrent real-time garbage collection. Slides from a talk at a Summer School on Trends in Concurrency, 2008.

[2] M. Herlihy and N. Shavit. The Art of Multiprocessor Programming, 2008.

- executes a CAS to append the new node; and
- executes a CAS to swing the queue's `tail` from the prior last node to the current last node.

Because these two steps are not executed atomically, every other method call must be prepared to encounter a half-finished `enqueue` call, and to finish the job ("help" the enqueuer).

In more detail, an enqueuer creates a new node with the new value to be enqueued, reads `tail`, and finds the node that appears to be last. To verify that node is indeed last, it checks whether the node has a successor. If the node does not have a successor, the thread attempts to append the new node using a CAS. If the CAS succeeds, the thread uses a second CAS to advance `tail` to the new node. Even if this second CAS fails, the thread can still return successfully because, as it happens, the CAS fails only if some other thread "helped" it by advancing `tail`. If the tail node has a successor, then the method tries to "help" other threads by advancing `tail` to refer directly to the successor before trying again to insert its own node.

The `dequeue` method checks that the queue is nonempty by checking that the next field of the head node is not null. It then executes a CAS to change `head` from the sentinel node to its successor, making the successor the new sentinel node. There is, however, a subtle issue: before advancing `head` one must make sure that `tail` is not left referring to the sentinel node which is about to be removed from the queue. To avoid this problem `dequeue` performs a test: if `head` equals `tail` and the (sentinel) node they refer to has a non-null `next` field, then the tail is deemed to be lagging behind. As in the `enqueue` method, `dequeue` then attempts to help make `tail` consistent by swinging it to the sentinel node's successor, and only then updates `head` to remove the sentinel. The value is read from the successor of the sentinel node.

As we have demonstrated above, Michael and Scott's queue implementation relies heavily on modification counters being shared between the allocator library and its client. Consequently, a linearizability approach to library abstraction cannot be used, since it requires the address space of components of the program to be completely disjoint.

Chapter 3

Preliminaries

In this chapter, we establish background for our contribution further in the paper. Section 3.1 defines in a common fashion the programming language and its operational semantics, which is defined w.r.t. machine states, primitive atomic commands and their semantics as state transformers. In Section 3.3, we introduce the Views Framework [2] as a general approach to abstracting from a machine state, axiomatising atomic commands and defining a composition operation.

3.1 Programming language

Syntax of the programming language. In our setting, the machines execute programs, which consist of several components. For simplicity of presentation, we further assume that there are two of them, which we call a *client* and a *library*, ranged by $\mathbb{C}$ and $\mathbb{L}$ correspondingly. Formally, a program $\mathbb{C}(\mathbb{L})$ is defined with the following notions:

$$
\begin{array}{ll}
\mathbb{P}, \mathbb{C}(\mathbb{L}) \in \mathsf{Prog} ::= \mathsf{let}\ \mathbb{L}\ \mathsf{in}\ \mathbb{C}; & \text{(programs)} \\
\mathbb{C} ::= C_1 \parallel \cdots \parallel C_N, \text{where } C_1, \ldots C_N \in \mathsf{Comm}; & \text{(client)} \\
\mathsf{Methods} = \{m_1, m_2, \ldots, m_k\} & \text{(methods names)} \\
C \in \mathsf{Comm} ::= \alpha \mid C\,;C \mid C + C \mid C^* \mid m(), \text{where } m \in \mathsf{Methods}; & \text{(commands)} \\
\mathbb{L} : \mathsf{Methods} \to \mathsf{Comm}; & \text{(library)}
\end{array}
$$

The grammar of syntactic commands includes atomic commands $\alpha \in \mathsf{Atom}$, sequential composition $C\,;C$, non-deterministic choice $C + C$, iteration C^*, which we consider to be finite, and method call $m()$ of a method $m \in \mathsf{Methods}$. A client $\mathbb{C}$ is a parallel composition of syntactic commands. A library $\mathbb{L}$ implements methods as syntactic commands, and together with $\mathbb{C}$ they form a program $\mathbb{C}(\mathbb{L})$. In a program $\mathbb{C}(\mathbb{L})$ we require that a library $\mathbb{L}$ provide implementation for all methods used by a client. For simplicity, we assume that clients do not have their own methods and that library methods do not call each other. Additionally, library methods do not

$\longrightarrow_{\mathbb{L}} : \mathsf{Comm} \times \Sigma \times \mathsf{Atom} \times \mathsf{Comm} \times \Sigma$:

$$\frac{\langle C_1, \sigma\rangle \xrightarrow{\alpha}_{\mathbb{L}} \langle C_1', \sigma'\rangle}{\langle C_1\,; C_2, \sigma\rangle \xrightarrow{\alpha}_{\mathbb{L}} \langle C_1'\,; C_2, \sigma'\rangle} \quad \frac{}{\langle C^*, \sigma\rangle \xrightarrow{\mathsf{skip}}_{\mathbb{L}} \langle C\,; C^*, \sigma\rangle} \quad \frac{}{\langle C^*, \sigma\rangle \xrightarrow{\mathsf{skip}}_{\mathbb{L}} \langle \mathsf{skip}, \sigma\rangle}$$

$$\frac{}{\langle \mathsf{skip}; C_2, \sigma\rangle \xrightarrow{\mathsf{skip}}_{\mathbb{L}} \langle C_2, \sigma\rangle} \quad \frac{}{\langle C_1 + C_2, \sigma\rangle \xrightarrow{\mathsf{skip}}_{\mathbb{L}} \langle C_1, \sigma\rangle} \quad \frac{}{\langle C_1 + C_2, \sigma\rangle \xrightarrow{\mathsf{skip}}_{\mathbb{L}} \langle C_2, \sigma\rangle}$$

$$\frac{}{\langle m(), \sigma\rangle \xrightarrow{\mathsf{call}_m}_{\mathbb{L}} \langle \mathbb{L}(m); \mathsf{ret}_m, \sigma\rangle} \quad \frac{\sigma_\alpha \in f_\alpha(\sigma)}{\langle \alpha, \sigma\rangle \xrightarrow{\alpha}_{\mathbb{L}} \langle \mathsf{skip}, \sigma_\alpha\rangle}$$

$\longrightarrow : \mathsf{Prog} \times \Sigma \times \mathsf{Atom} \times \mathsf{Prog} \times \Sigma$:

$$\frac{\langle C_i, \sigma\rangle \xrightarrow{\alpha}_{\mathbb{L}} \langle C_i', \sigma'\rangle \quad 1 \le i \le \mathsf{NThreads}}{\langle \mathsf{let}\ \mathbb{L}\ \mathsf{in}\ C_1 \parallel \cdots \parallel C_i \parallel \cdots \parallel C_{\mathsf{NThreads}}, \sigma\rangle \xrightarrow{\alpha} \langle \mathsf{let}\ \mathbb{L}\ \mathsf{in}\ C_1 \parallel \cdots \parallel C_i' \parallel \cdots \parallel C_{\mathsf{NThreads}}, \sigma'\rangle}$$

Figure 3.1: The operational semantics of programs

have parameters and return values, which is explained by the fact that a client and a library share all the memory.

In order to simplify presentation of the operational semantics, we assume that the set of atomic commands Atom can be split onto sets $\mathsf{Atom_C}$ and $\mathsf{Atom_L}$ consisting of commands performed by client and library. Their common part $\mathsf{Atom_C} \cap \mathsf{Atom_L} = \mathsf{Atom_I} \uplus \{\mathsf{skip}\}$ consists of atomic commands call_m and ret_m for each $m \in \mathsf{Methods}$, which are called *interface* commands, and also a special identity command skip. Syntactic commands $C_1, \ldots, C_N$ of a client $\mathbb{C}$ are required to use only non-interface atomic commands from $\mathsf{Atom_C}$, and similarly each method m of a library $\mathbb{L}$ uses only non-interface commands from $\mathsf{Atom_L}$ in $\mathbb{L}(m)$.

Operational semantics. Programs operate with machine states from the set Σ, ranged over by σ. The interpretation for each atomic command α is given by non-deterministic state transformer $f_\alpha : \Sigma \to \mathcal{P}(\Sigma)$ and Func is a set of all them. Where necessary, we lift non-deterministic state transformers to sets of states: for $S \in \mathcal{P}(\Sigma)$, $f_\alpha(S) = \bigcup\{f_\alpha(\sigma) \mid \sigma \in S\}$. We assume that the atomic command skip has the interpretation $f_{\mathsf{skip}} = \lambda\sigma.\,\{\sigma\}$, and let $f_{\mathsf{call}_m} = f_{\mathsf{ret}_m} = f_{\mathsf{skip}}$.

Additionally, we require that Atom be closed under composition of functions: for all $\alpha, \beta \in \mathsf{Atom}$ there is $\gamma \in \mathsf{Atom}$, written $\gamma = \alpha \circ \beta$, such that $f_\gamma = (f_\alpha \circ f_\beta)$. In cases when a command α is composed with skip, we let $\alpha \circ \mathsf{skip} = \mathsf{skip} \circ \alpha = \alpha$.

Semantics of programs is defined by the transition relation $\longrightarrow : \mathsf{Prog} \times \Sigma \times \mathsf{Atom} \times \mathsf{Prog} \times \Sigma$, which is introduced Figure 3.1. The relation $\longrightarrow$ labels transition between programs and their machine states with atomic commands performed. We also define a *multiple-step transition relation* $\Longrightarrow : \mathsf{Prog} \times \Sigma \times \mathsf{Atom} \times \mathsf{Prog} \times \Sigma$ with the following rules:

- $\langle \mathbb{P}, \sigma \rangle \stackrel{\mathsf{skip}}{\Longrightarrow} \langle \mathbb{P}, \sigma \rangle$;
- if $\langle \mathbb{P}, \sigma \rangle \stackrel{\alpha}{\longrightarrow} \langle \mathbb{P}', \sigma' \rangle$, then $\langle \mathbb{P}, \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \mathbb{P}', \sigma' \rangle$;
- if $\langle \mathbb{P}, \sigma \rangle \stackrel{\alpha}{\longrightarrow} \langle \mathbb{P}', \sigma' \rangle$, $\langle \mathbb{P}', \sigma' \rangle \stackrel{\alpha'}{\longrightarrow} \langle \mathbb{P}'', \sigma'' \rangle$ and either $\alpha, \alpha' \in \mathsf{Atom_C} \setminus \mathsf{Atom_I}$ or $\alpha, \alpha' \in \mathsf{Atom_L} \setminus \mathsf{Atom_I}$, then $\langle \mathbb{P}, \sigma \rangle \stackrel{\alpha' \circ \alpha}{\Longrightarrow} \langle \mathbb{P}'', \sigma'' \rangle$;
- if $\langle \mathbb{P}, \sigma \rangle \stackrel{\alpha}{\longrightarrow} \langle \mathbb{P}', \sigma' \rangle$, $\langle \mathbb{P}', \sigma' \rangle \stackrel{\alpha'}{\longrightarrow} \langle \mathbb{P}'', \sigma'' \rangle$ and either $\alpha = \mathsf{skip} \wedge \alpha' \in \mathsf{Atom_I}$ or $\alpha' = \mathsf{skip} \wedge \alpha \in \mathsf{Atom_I}$, then $\langle \mathbb{P}, \sigma \rangle \stackrel{\alpha' \circ \alpha}{\Longrightarrow} \langle \mathbb{P}'', \sigma'' \rangle$;

3.2 Transition traces semantics

Giving a semantics for concurrent programs is notoriously difficult, because of a inevitable duality to be resolved, which is between supporting compositional reasoning about programs and capturing all operational behaviours of them. After Brookes in [1], we make use *transition traces* of programs, which are generated with assumptions of any possible interference with environment capable of changing machine state in an arbitrary way.

Informally, a transition trace of a program $\mathbb{P}$ is defined to be a finite sequence $(\alpha_1, \sigma_1, \sigma_1')(\alpha_2, \sigma_2, \sigma_2') \ldots (\alpha_k, \sigma_k, \sigma_k')$ such that $\mathbb{P}$ performs a computation from the state σ_1 to the state σ_k' having been interrupted by environment exactly $k-1$ times. For this trace, environment did transitions between pairs of states σ_i' and σ_{i+1} $(1 \leq i < k)$. The formal definition follows:

Definition 1 (Transition trace)**.** *A transition trace of a program $\mathbb{P}$ is such sequence $Z \in \mathsf{Traces} = (\mathsf{Atom} \times \Sigma \times \Sigma)^*$ of the form $Z = (\alpha_1, \sigma_1, \sigma_1')(\alpha_2, \sigma_2, \sigma_2') \ldots (\alpha_k, \sigma_k, \sigma_k')$ that the following holds:*

$$\exists \mathbb{P}_1, \mathbb{P}_2, \ldots \mathbb{P}_{k+1}. \mathbb{P} = \mathbb{P}_1 \wedge \langle \mathbb{P}_1, \sigma_1 \rangle \stackrel{\alpha_1}{\Longrightarrow} \langle \mathbb{P}_2, \sigma_1' \rangle \wedge \langle \mathbb{P}_2, \sigma_2 \rangle \stackrel{\alpha_2}{\Longrightarrow} \langle \mathbb{P}_3, \sigma_2' \rangle \wedge \ldots \wedge \langle \mathbb{P}_k, \sigma_k \rangle \stackrel{\alpha_k}{\Longrightarrow} \langle \mathbb{P}_{k+1}, \sigma_k' \rangle,$$

and the set of all transition traces of program $\mathbb{P}$ is then denoted by $\mathcal{T}[\![\mathbb{P}]\!]$.

3.3 Views

Proving properties of transition traces of a program requires considering an arbitrary environment. However, on practice, programs are usually designed with a specific way of communication between their components, which means that intended environment of such component is not arbitrary – there is a contract between them to follow. In order to be able to express such contracts, we use the Views Framework of Dinsdale-Young et al [2], which generalises reasoning systems for concurrent programs.

Typically, reasoning systems do not require the user to reason directly about the state; they provide an abstract representation of the state that supports a particular

form of reasoning. Abstractions may contain some extra information that does not exist on the level of concrete machine states, but helps establishing properties of them. This information may serve as a contract between a program and its environment, which may be formalised in terms of an ownership of certain memory locations or a protocol of performing specific changes to the shared memory. The Views Framework [2] offers a general way of introducing the notion of an abstract state, which with additional requirements can be instantiated into many existing program logics.

The Views Framework includes the following parameters that we operate with:

- A commutative *View monoid* $(\mathsf{View}, *, u)$, which is an algebraic structure on a set View of abstract states, ranged over by p, with an associative and commutative operation $*$ that is called *view composition*, and also a unit u.
- A *reification* function $\lfloor - \rfloor : \mathsf{View} \to \mathcal{P}(\Sigma)$, which maps views to corresponding sets of concrete state.
- A set of axioms from $\mathsf{View} \times \mathsf{Atom} \times \mathsf{View}$, ranged over by triples $\alpha \Vdash \{p\}\{q\}$.

A View monoid introduces *views* to the framework as abstract states, which are related to machine states by means of reification. The composition $p_1 * p_2$ embodies combining properties specified by two views p_1 and p_2. Although composition is a total operation, the view $p_1 * p_2$ may not have any corresponding machine state, i.e. $\lfloor p_1 * p_2 \rfloor = \emptyset$, which is the case when contradictory properties are combined. A unit u is a view that represents no knowledge about machine states, and thus $\lfloor p * u \rfloor = \lfloor u * p \rfloor = \lfloor p \rfloor$ for all $p \in \mathsf{View}$.

Axioms of the framework are required to satisfy the following constraint:

$$\alpha \Vdash \{p\}\{q\} \implies \forall r \in \mathsf{View}.\, f_\alpha(\lfloor p * r \rfloor) \subseteq \lfloor q * r \rfloor \tag{3.1}$$

Two things are checked in the formula above. The first is that $\alpha \Vdash \{p\}\{q\}$ ensures that making a transition with a transformer α is allowed in an abstract state p, which is equivalent to checking if $f_\alpha(\lfloor p \rfloor) \subseteq \lfloor q \rfloor$. The other requirement of $\alpha \Vdash \{p\}\{q\}$ is that any view r, which a program's environment could possibly have, remains unchanged, and that is formalised by $\forall r \in \mathsf{View}.\, f_\alpha(\lfloor p * r \rfloor) \subseteq \lfloor q * r \rfloor$. Since there always is a unit $u \in \mathsf{View}$, the first requirement is just an instance of the second. Also, it is easy to see that $\alpha \Vdash \{p * r\}\{q * r\}$ holds whenever $\alpha \Vdash \{p\}\{q\}$ does, and further we call this observation the *Frame Property of axioms*.

We require the Framework to provide axioms with the strongest postconditions in the way presented below:

Definition 2 (The Strongest Postconditions)**.** *Given $\alpha \in \mathsf{Atom}$ and $p \in \mathsf{View}$, we define the set of strongest postconditions $\mathsf{SP}(\alpha, q)$:*

$$\mathsf{SP}(\alpha, p) = \{q \mid \alpha \Vdash \{p\}\{q\} \wedge \forall q'.\, \alpha \Vdash \{p\}\{q'\} \implies q \subseteq q'\},$$

and say that (p, α, q) is best axiom when $q \in \mathsf{SP}(\alpha, p)$.

Definition 3. *We say that the Views Framework is supplied with best axioms, if:*

$$\forall \alpha, p.\, (\exists q.\, \alpha \Vdash \{p\}\{q\}) \implies \mathsf{SP}(p, \alpha) \neq \emptyset.$$

As stated in the definition above, whenever there is an axiom (p, α, q) for a transformer α, there must also be best axiom for it. In the rest of the work we assume that this is always the case.

Another adjustment to the Views Framework that we introduce is aimed at a reification function: in our work views represent information about the whole machine state, and **memory locations from it are never being changed**. We achieve this by requiring reification to satisfy the following property:

Property 4. $\lfloor p * q \rfloor \subseteq \lfloor p \rfloor \cap \lfloor q \rfloor$.

This property implicitly states two facts about $p * q$: memory locations of states from $\lfloor p \rfloor, \lfloor q \rfloor$ and $\lfloor p * q \rfloor$ are the same, and also $p * q$ represents no more machine states than p or q do, because it contains more restrictions to them.

Chapter 4

Safety and local semantics

In this chapter we combine views and transition traces into formalisms that represent a contract between a program and its external environment.

Consider a program $\mathbb{C}(\mathbb{L})$ that consists of a client $\mathbb{C}$ and a library $\mathbb{L}$. To formulate our result, we need to give a semantics to parts of $\mathbb{C}(\mathbb{L})$: the library $\mathbb{L}$ considered in isolation from its client and the client $\mathbb{C}$ considered in isolation from the implementation of the library it uses. A semantics we would like to use is a subset of transition traces semantics. In this chapter we introduce notions of safety for programs, safe semantics and then explain how to extend them for clients and libraries separately.

We consider a program $\mathbb{P}$ being specified with a view p, which serves as an initial abstract state, i.e. a precondition. An unknown environment of $\mathbb{P}$ is expected to have its own view as an abstraction of a memory state. Along the execution of $\mathbb{P}$, views of a program and environment are changed, however, the contract between $\mathbb{P}$ and its environment requires that their views soundly describe a machine state after each transition. Safety is a property of programs that incorporates this intuition.

Definition 5 (Safety)**.** *For a command* $\mathbb{P} \in \mathsf{Prog}$ *and view* $p \in \mathsf{View}$*, the safety judgement* safe *is defined to be the maximal relation over* $\mathsf{Prog} \times \mathsf{View} \times \mathsf{View}$ *such that* $\mathsf{safe}(\mathbb{P}, p)$ *holds if and only if the following is satisfied:*

$$\forall \alpha, \mathbb{P}', \sigma, \sigma' . \sigma \in \lfloor p \rfloor \wedge \langle \mathbb{P}, \sigma \rangle \overset{\alpha}{\Longrightarrow} \langle \mathbb{P}', \sigma' \rangle \implies \exists p' . p' \in \mathsf{SP}(\alpha, p) \wedge \mathsf{safe}(\mathbb{P}', p'). \quad (4.1)$$

Safety of $\mathbb{P}$ w.r.t. p considers each transition from $\mathbb{P}$ in a state σ satisfying p that is enabled by operational semantics, and ensures that there is an axiom (p, α, p') in the framework and the same is true for further transitions of this kind. Being able to choose an axiom means that $\sigma' \in \lfloor p' \rfloor$ holds of the resulting machine state $\sigma' \in f_\alpha(\sigma)$ of a transition, because $\sigma \in \lfloor p \rfloor$ and according to the axiom, $f_\alpha(\sigma) \subseteq f_\alpha(\lfloor p \rfloor) \subseteq \lfloor p' \rfloor$.

Intuitively, $\mathsf{safe}(\mathbb{P}, p)$ can be understood as picking views consistent with machine states along traces of $\mathbb{P}$, which means absence of memory faults in these traces w.r.t. precondition p. Note that only transition traces are checked in safety: $\mathsf{safe}(\mathbb{P}', p')$ considers every possible machine state $\sigma'' \in \lfloor p' \rfloor$ as initial, thus, when $\mathsf{safe}(\mathbb{P}, p)$ holds, between any transition $\langle \mathbb{P}, \sigma \rangle \overset{\cdot}{\Longrightarrow} \langle \mathbb{P}', \sigma' \rangle$ and $\langle \mathbb{P}', \sigma'' \rangle \overset{\cdot}{\Longrightarrow} \langle \_, \_ \rangle$ the possibility for environment to change a state between program's transitions from σ' to σ''

is considered. Environment is restricted in changes he can make to a machine state, because his state change must not invalidate program's view. This is exactly $\mathsf{safe}(\mathbb{P}, p)$ formalises a contract with environment.

The set of all traces informally mentioned above is further defined as a semantics for programs. Analogously to Traces, we let $\mathsf{Traces}^{\#} = (\mathsf{Atom} \times \mathsf{View} \times \mathsf{View})^*$, ranged over by $Z^{\#}$, be the set of all possible *abstract traces*. We use ε and $\varepsilon^{\#}$ to represent an empty trace and abstract trace respectively.

Definition 6 (Safe semantics)**.** *Let* $\mathcal{S}[\![-]\!] : \mathsf{Comm} \times \mathsf{View} \to \mathsf{Traces} \times \mathsf{Traces}^{\#}$ *be the minimal function satisfying constraint below with subset inclusion ordering lifted to* $\mathsf{Comm} \times \mathsf{View} \to \mathsf{Traces} \times \mathsf{Traces}^{\#}$ *pointwise:*

$$\mathcal{S}[\![\mathbb{P}]\!](p) \triangleq \{(\alpha, \sigma, \sigma')Z, (\alpha, p, p')Z^{\#} \mid \exists \mathbb{P}'.\, \sigma \in \lfloor p \rfloor \wedge \langle \mathbb{P}, \sigma \rangle \xRightarrow{\alpha} \langle \mathbb{P}', \sigma' \rangle \wedge p' \in \mathsf{SP}(\alpha, p) \wedge (Z, Z^{\#}) \in \mathcal{S}[\![\mathbb{P}']\!](p')\} \cup \{(\varepsilon, \varepsilon^{\#})\}.$$

When $\mathsf{safe}(\mathbb{P}, p)$ *holds,* $\mathcal{S}[\![\mathbb{P}]\!](p)$ *is called the safe semantics of* $\mathbb{P}$ *w.r.t.* p*.*

In definitions of both safety and safe semantics, the strongest views are picked. Such a choice is substantial for semantics, because otherwise abstract state may be weakened too much, which is indeed sound, but imposes less restrictions on the environment than program may actually rely on. Also, analogously to the safety definition, $\sigma' \in \lfloor p' \rfloor$ in the set comprehension above.

Characterisation with traces. Definition 6 presents safe semantics inductively, which illustrates its relation to coinductive definition of safety, but is hard to reason about on practice. To make it more intuitive, in Proposition 8 we move on to a definition that checks properties of transition traces explicitly.

For this purpose we define a *concretisation* of an abstract trace, which effectively lifts a reification from views to abstract traces pointwise and checks that $\sigma \in \lfloor p \rfloor$ holds of each concrete state σ from a trace Z and a corresponding view p from an abstract trace $Z^{\#}$. In *validity* of an abstract trace, we ensure that its views are picked according to best axioms.

Definition 7. *Given an abstract trace* $Z^{\#} = \{(\alpha'_i, p_i, p_{i+1})\}_{i=1}^{k}$ *(*$k \geq 0$*), we say that:*

- $\gamma(Z^{\#}) \subseteq \mathsf{Traces}$ *is a set of concretisations of* $Z^{\#}$*, when each trace* $Z \in \gamma(Z^{\#})$ *is of the form* $Z = \{(\alpha_i, \sigma_i, \sigma'_i)\}_{i=1}^{k}$ *and satisfies the requirement:*

$$\forall i.\, 1 \leq i \leq k \implies \alpha_i = \alpha'_i \wedge \sigma_i \in \lfloor p_i \rfloor \wedge \sigma'_i \in \lfloor p'_i \rfloor;$$

- $Z^{\#}$ *is valid, written* $\mathsf{valid}(Z^{\#})$*, if* $\forall i.\, 1 \leq i \leq k \implies p_{i+1} \in \mathsf{SP}(\alpha_i, p_i)$*.*

Proposition 8. *If* $\mathsf{safe}(\mathbb{P}, p)$ *holds, then safe semantics may be represented as a set of pairs* $(Z, Z^{\#})$ *of valid abstract traces* $Z^{\#}$ *and their concretisations* Z*:*

$$\mathcal{S}[\![\mathbb{P}]\!](p) = \{(Z, Z^{\#}) \mid Z \in \mathcal{T}[\![\mathbb{P}]\!] \wedge Z^{\#} = (\_, p, \_)\_ \in \mathsf{Traces}^{\#} \wedge Z \in \gamma(Z^{\#}) \wedge \mathsf{valid}(Z^{\#})\}.$$

Library-local semantics. To give a semantics to a library $\mathbb{L}$, we consider specific clients of it, which are programs $\{\mathsf{MGC}_n\}_{n\geq 1}$ that are parametrised by the number of threads n. In each its thread MGC_n calls an arbitrary method m of a library $\mathbb{L}$ on each step of execution. More formally, these clients of $\mathbb{L}$ are:

$$\mathsf{MGC}_n(\mathbb{L}) = \text{let } \mathbb{L} \text{ in } C_1^{\mathsf{mgc}} \parallel C_2^{\mathsf{mgc}} \parallel \cdots \parallel C_n^{\mathsf{mgc}},$$
$$\text{where } C_i^{\mathsf{mgc}} = (m_1 + m_2 + \cdots + m_k)^* \text{ and } \mathsf{dom}(\mathbb{L}) = \{m_1, m_2, \ldots, m_k\}$$

The safe library-local semantics of $\mathbb{L}$ and its safety are then defined respectively as

$$\mathcal{S}[\![\mathbb{L}]\!](p) \triangleq \bigcup_{n\geq 1} \mathcal{S}[\![\mathsf{MGC}_n(\mathbb{L})]\!](p) \text{ and } \mathsf{safe}(\mathbb{L}, p) \triangleq \bigwedge_{n\geq 1} \mathsf{safe}(\mathsf{MGC}_n(\mathbb{L}), p).$$

Internal safety. We further present the notion of internal safety of programs that consist of two components.

Definition 9 (Internal safety)**.** *For a program $\mathbb{C}(\mathbb{L})$ and views $p_c, p_l \in \mathsf{View}$, $\mathsf{safeint}$ is defined to be the maximal relation such that judgement $\mathsf{safeint}(\mathbb{C}(\mathbb{L}), p_c, p_l)$ holds if and only if the following condition is satisfied:*

$$\forall \alpha, \mathbb{P}, \sigma, \sigma'.\, \sigma \in \lfloor p_c * p_l \rfloor \wedge \langle \mathbb{C}(\mathbb{L}), \sigma\rangle \xRightarrow{\alpha} \langle \mathbb{P}, \sigma'\rangle \implies$$
$$(\alpha \in \mathsf{Atom_C} \implies \exists p_c'.\, p_c' \in \mathsf{SP}(\alpha, p_c) \wedge \mathsf{safeint}(\mathbb{P}, p_c', p_l)) \wedge$$
$$(\alpha \in \mathsf{Atom_L} \implies \exists p_l'.\, p_l' \in \mathsf{SP}(\alpha, p_l) \wedge \mathsf{safeint}(\mathbb{P}, p_c, p_l')).$$

Analogously to safety, internal safety of $\mathbb{C}(\mathbb{L})$ w.r.t. $p_c * p_l$ considers every possible transition from $\mathbb{C}(\mathbb{L})$ in a state $\sigma \in \lfloor p_c * p_l \rfloor$ and chooses an axiom (p_c, α, p_c') or (p_l, α, p_l') that justifies the transition and enables internal safety of $\mathsf{safeint}(\mathbb{P}, p_c', p_l)$ or $\mathsf{safeint}(P', p_c, p_l')$ respectively. It is easy to show that σ' satisfies $p_c' * p_l$ or $p_c * p_l'$ in either case. The choice of an axiom depends on the component that performed a transition, and this is what differs internal safety from safety: here an axiom is picked w.r.t. to only a part of the abstract state $p_c * p_l$ that belongs to the component, and the Frame Property of axioms allows to conclude that a view of the other component remains holding. Consequently, internal safety explicitly states that a client $\mathbb{C}$ and a library $\mathbb{L}$ are changing machine states with respect to their own abstract states p_c and p_l.

Similarly to the safe semantics of $\mathbb{C}(\mathbb{L})$, one can think of picking axioms along transition traces of $\mathbb{C}(\mathbb{L})$ in a way required by internal safety and define a set of internally safe traces, or a *internally safe semantics* $\mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$ w.r.t. to local views p_c and p_l. We omit an inductive definition of $\mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$ and present a straightforward one.

Definition 10. *Given a pair of valid abstract traces $X^\# = \{(\alpha_i, p_i, p_{i+1})\}_{i=1}^k$ and $Y^\# = \{(\alpha_i', p_i', p_{i+1}')\}_{i=1}^k$ $(k \geq 0)$, we define the result of the operation $X^\# \circledast Y^\#$ to be an abstract trace $Z^\# = \{(\alpha_i'', p_i * p_i', p_{i+1} * p_{i+1}')\}_{i=0}^k$ if the following requirement is met:*

$$\forall i.\, 1 \leq i \leq n \implies (\alpha_i = \alpha_i' = \alpha_i'' \in \mathsf{Atom_I}) \vee$$
$$(\alpha_i = \mathsf{skip} \wedge \alpha_i'' = \alpha_i' \in \mathsf{Atom_L}) \vee (\alpha_i' = \mathsf{skip} \wedge \alpha_i'' = \alpha_i \in \mathsf{Atom_C}).$$

The definition of $\circledast$ lifts the operation $*$ from views to valid abstract traces pointwise, but only when an additional requirement on commands is satisfied. The meaning of it is to require that an abstract trace $X^\#$ does not change its view, when so does $Y^\#$ and vice versa.

Definition 11 (Internally safe traces)**.** *For a program* $\mathbb{C}(\mathbb{L})$ *and views* $p_c, p_l \in \mathsf{View}$*, the set of internally safe traces is defined as follows:*

$$\mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l) \triangleq \{(Z, X^\#, Y^\#) \mid Z \in \mathcal{T}[\![\mathbb{C}(\mathbb{L})]\!] \wedge X^\# = (\_, p_c, \_)\,\_ \wedge \\ Y^\# = (\_, p_l, \_)\,\_ \wedge \mathsf{valid}(X^\#) \wedge \mathsf{valid}(Y^\#) \wedge Z \in \gamma(X^\# \circledast Y^\#)\}$$

The difference in representation of internally safe traces and safe traces is that for the former we maintain two *local* abstract traces $X^\#$ and $Y^\#$ corresponding to view changes by a client $\mathbb{C}$ and a library $\mathbb{L}$. Whenever either of them performs a transition, this is reflected in its local abstract trace, and for convenience of representation, another component is assumed to stutter.

It is important that strongest views in abstract traces $X^\#$ and $Y^\#$ are chosen independently. When a view $p_c' \in \mathsf{SP}(\alpha, p_c)$ is chosen for a client's transition in $X^\#$ and p_l is a library's view, a machine state σ' that is a result of a transition of $\mathbb{C}(\mathbb{L})$ is checked against $p_c' * p_l$. Although $\alpha \Vdash \{p_c * p_l\}\{p_c' * p_l\}$ holds by the Frame Property, in general case $p_c' * p_l \in \mathsf{SP}(\alpha, p_c * p_l)$ may not hold. Thus, total views of $\mathbb{C}(\mathbb{L})$ in internally safe semantics are not strongest, unlike views picked in safe semantics.

Chapter 5

Library abstraction

This chapter is dedicated to the definition of library abstraction and a justification of its soundness, which is the Abstraction Theorem.

Projections of traces. Consider a trace $X \in \mathcal{T}[\![\mathbb{C}(\mathbb{L})]\!]$ of a program $\mathbb{C}(\mathbb{L})$. We introduce $\mathsf{client}(X)$ and $\mathsf{lib}(X)$, which for a given trace return only those transitions from it that are performed by client and library respectively (call and return transitions are considered to belong to both). Here follows the definition of client (the definition of lib is analogous):

$$\mathsf{client}((\alpha, \sigma, \sigma')\, Z) = \begin{cases} (\alpha, \sigma, \sigma')\, \mathsf{client}(Z), & \text{if } \alpha \in \mathsf{Atom_C}; \\ \mathsf{client}(Z), & \text{otherwise} \end{cases}$$
$$\mathsf{client}(\varepsilon) = \varepsilon.$$

We also define another projection: $\mathsf{states} : \mathsf{Traces} \to (\Sigma \times \Sigma)^*$ that omits atomic commands from each transition in a trace, $\mathsf{views} : \mathsf{Traces}^{\#} \to (\mathsf{View} \times \mathsf{View})^*$ that does the same to abstract traces, and $\mathsf{interface} : \mathsf{Traces} \to \mathsf{Atom_I}^*$ that filters out everything but interface actions:

$$\mathsf{states}(\varepsilon) = \varepsilon$$
$$\mathsf{states}((\_, \sigma, \sigma')\, X) = (\sigma, \sigma')\, \mathsf{states}(X)$$

$$\mathsf{views}(\varepsilon^{\#}) = \varepsilon$$
$$\mathsf{views}((\_, p, p')\, X^{\#}) = (p, p')\, \mathsf{views}(X^{\#})$$

$$\mathsf{interface}((\alpha, \_, \_)\, Z) = \begin{cases} \alpha\, \mathsf{interface}(Z), & \text{if } \alpha \in \mathsf{Atom_I} \\ \mathsf{interface}(Z), & \text{otherwise} \end{cases}$$
$$\mathsf{interface}(\varepsilon) = \varepsilon.$$

Library abstraction. Let us define a relation $\preceq \subseteq (\mathsf{View} \times \mathsf{View})^*$:

$$V_1 \preceq V_2 \triangleq V_1 = V_2 = \varepsilon \vee \exists n, p_1, \ldots, p_{n+1}, p'_1, \ldots, p'_{n+1}.\, n \geq 1 \wedge$$
$$V_1 = (p_1, p_2)(p_2, p_3) \ldots (p_n, p_{n+1}) \wedge V_2 = (p'_1, p'_2)(p'_2, p'_3) \ldots (p'_n, p'_{n+1}) \wedge$$
$$\forall i.\, 1 \leq i \leq n+1 \implies \mathsf{skip} \Vdash \{p_i\}\{p'_i\}.$$

We define the library abstraction relation, which states that for each pair of concrete and abstract traces of $\mathbb{L}_1$ it is possible to find a corresponded pair of traces of $\mathbb{L}_2$ with a weaker specification and identical behaviour on a machine level.

Definition 12. *The* ***library abstraction*** *is a binary relation* $\sqsubseteq$ *on libraries that is parametrised by* $p \in \mathsf{View}$ *and is defined as follows:*

$$\mathbb{L}_1 \sqsubseteq_p \mathbb{L}_2 \triangleq \mathsf{safe}(\mathbb{L}_1, p) \land \mathsf{safe}(\mathbb{L}_2, p) \land \forall (Y_1, Y_1^\#) \in \mathcal{S}[\![\mathbb{L}_1]\!](p). \\ \exists (Y_2, Y_2^\#) \in \mathcal{S}[\![\mathbb{L}_2]\!](p). (Y_1, Y_1^\#) \sqsubseteq (Y_2, Y_2^\#).$$

where $(Y_1, Y_1^\#) \sqsubseteq (Y_2, Y_2^\#) \triangleq \mathsf{states}(Y_1) = \mathsf{states}(Y_2) \land \mathsf{views}(Y_1^\#) \preceq \mathsf{views}(Y_2^\#) \land \mathsf{interface}(Y_1) = \mathsf{interface}(Y_2)$.

We now justify that the notion of library abstraction is sound by establishing the Abstraction Theorem that justifies abstracting from an implementation of a library with its specification while reasoning about its client. The theorem states that internally safe executions of a client of a concurrent library are still reproducible with another library, provided that libraries are in the library abstraction relation.

Theorem 13 (Abstraction)**.** *If libraries* $\mathbb{L}_1$ *and* $\mathbb{L}_2$ *are such that* $\mathbb{L}_1 \sqsubseteq_{p_l} \mathbb{L}_2$ *hold, then:*

$$\forall (Z_1, X_1^\#, Y_1^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L}_1)]\!](p_c, p_l). \exists (Z_2, X_2^\#, Y_2^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L}_2)]\!](p_c, p_l). \\ \mathsf{client}(Z_1) = \mathsf{client}(Z_2) \land \mathsf{states}(\mathsf{lib}(Z_1)) = \mathsf{states}(\mathsf{lib}(Z_2)) \land \\ X_1^\# = X_2^\# \land Y_1^\# \preceq Y_2^\#.$$

The Abstraction Theorem establishes a correlation between internally safe traces of $\mathbb{C}(\mathbb{L}_1)$ and $\mathbb{C}(\mathbb{L}_2)$. However, while satisfying the purpose of specifying the contract between the client and the library, in order to do so internally safe traces exhibit details of interactions between the components, which might be redundant on practice. We further focus on encapsulating the required information about the contract in the internal safety and provide similar results for less restrictive semantics.

In order to present the first corollary of the Abstraction Theorem, we have to require more properties from axioms of the Views Framework.

Definition 14. *We say that the set of best axioms of the Views Framework is closed under Frame property, when the following holds:*

$$\forall p, q, \alpha. q \in \mathsf{SP}(\alpha, p) \implies q * r \in \mathsf{SP}(\alpha, p * r).$$

The property above strengthens the Frame Property of the axioms by requiring each best axiom (p, α, q) remain the best even if the view p is extended with another view r.

Definition 15. *We say that the axioms of the Views Framework are generated from semantics, if the following holds:*

$$\alpha \Vdash \{p\}\{q\} \iff \forall r \in \mathsf{View}. f_\alpha(\lfloor p * r \rfloor) \subseteq \lfloor q * r \rfloor$$

Lemma 16. *When best axioms of the View Framework are closed under Frame property and are generated from semantics, given* $\mathsf{safeint}(\mathbb{C}(\mathbb{L}), p_c, p_l)$*, any abstract trace* $Z^\#$ *of a trace* Z *can be overapproximated by local abstract traces* $X^\#$ *and* $Y^\#$*:*

$$\mathsf{safeint}(\mathbb{C}(\mathbb{L}), p_c, p_l) \implies \forall Z, Z^\#. (Z, Z^\#) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l) \implies \\ \exists X^\#, Y^\#. (Z, X^\#, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l) \wedge \mathsf{views}(Z^\#) \preceq \mathsf{views}(X^\# \circledast Y^\#)$$

Lemma 17. *When best axioms of the View Framework are closed under Frame property and are generated from semantics,*

$$\forall Z, X^\#, Y^\#. (Z, X^\#, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l) \implies (Z, X^\# \circledast Y^\#) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l).$$

By using Lemmas 16 and 17 respectively before and after Theorem 13, we obtain the result, in which correlation between safe traces is established.

Corollary 18. *Assume that best axioms of the Views Framework are closed under Frame property. If a client* $\mathbb{C}$ *and libraries* $\mathbb{L}_1$ *and* $\mathbb{L}_2$ *are such that* $\mathsf{safeint}(\mathbb{C}(\mathbb{L}_1), p_c, p_l)$ *and* $\mathbb{L}_1 \sqsubseteq_{p_l} \mathbb{L}_2$ *hold, then:*

$$\forall (Z_1, Z_1^\#) \in \mathcal{S}[\![C(L_1)]\!](p_c * p_l). \exists (Z_2, Z_2^\#) \in \mathcal{S}[\![C(L_2)]\!](p_c * p_l). Z_1^\# \preceq Z_2^\# \wedge \\ \mathsf{client}(Z_1) = \mathsf{client}(Z_2) \wedge \mathsf{states}(\mathsf{lib}(Z_1)) = \mathsf{states}(\mathsf{lib}(Z_2)).$$

Relating safe traces instead of internally safe ones allows to hide details of checking the contract between components in semantics and encapsulate it into $\mathsf{safeint}(\mathbb{C}(\mathbb{L}_1), p_c, p_l)$. This result might be of use in order to apply the Abstraction Theorem multiple times to programs consisting of more than two components. While in the present work the application of Corollary 18 is restricted by the programming language design, we intend to explore this direction in the future work.

The other consequence of the Abstraction Theorem is formulated for traces, in the setting when the program executes in the absence of the environment. This means that in transition traces of a closed program, states are not changed between two consequent transitions no external environment is present in the execution, which we formally formulate in the definition of closed semantics:

Definition 19. *Closed semantics* $\mathcal{S}_{\mathsf{closed}}[\![\mathbb{P}]\!](p)$ *of a program* $\mathbb{P}$ *w.r.t. a view* p *is:*

$$\mathcal{S}_{\mathsf{closed}}[\![\mathbb{P}]\!](p) = \{Z \mid Z \in \mathcal{T}[\![\mathbb{P}]\!] \wedge \exists n, \{\alpha_i\}_{i=1}^n, \{\sigma_i\}_{i=1}^{n+1}. \sigma_1 \in \lfloor p \rfloor \wedge \\ Z = (\alpha_1, \sigma_1, \sigma_2)(\alpha_2, \sigma_2, \sigma_3) \ldots (\alpha_n, \sigma_n, \sigma_{n+1})\} \cup \{\varepsilon\}$$

In this case we are able to establish a correspondence between traces of closed semantics of $\mathbb{C}(\mathbb{L}_1)$ and $\mathbb{C}(\mathbb{L}_2)$, which are effectively standard traces.

Corollary 20. *If a client* $\mathbb{C}$ *and libraries* $\mathbb{L}_1$ *and* $\mathbb{L}_2$ *are such that* $\mathsf{safeint}(\mathbb{C}(\mathbb{L}_1), p_c, p_l)$ *and* $\mathbb{L}_1 \sqsubseteq_{p_l} \mathbb{L}_2$ *hold, then:*

$$\forall Z_1 \in \mathcal{S}_{\mathsf{closed}}[\![\mathbb{C}(\mathbb{L}_1)]\!](p_c * p_l). \exists Z_2 \in \mathcal{S}_{\mathsf{closed}}[\![\mathbb{C}(\mathbb{L}_2)]\!](p_c * p_l). \mathsf{client}(Z_1) = \mathsf{client}(Z_2) \wedge \\ \mathsf{states}(\mathsf{lib}(Z_1)) = \mathsf{states}(\mathsf{lib}(Z_2)).$$

Knowing that there is no environment in the execution allows to hide all information about contracts from semantics. On practice this allows proving properties of components interacting by means of shared memory, and applying standard methods like linearizability to prove properties of the other components, which do not share memory with each other.

5.1 Proof outline

Stuttering and skip-mumbling. We present closure properties of safe semantics that facilitate the proof of the Abstraction theorem.

Definition 21 (Stuttering)**.** *A set* $S \subseteq \mathsf{Traces} \times \mathsf{Traces}^{\#}$ *is closed under stuttering, when for every trace* $(X\,Y, X^{\#}\,Y^{\#}) \in S$*, view* p' *and state* σ *such that* $\sigma \in \lfloor p' \rfloor$ *and* p' *is such a view that* $X^{\#} = \_(\_, \_, p')$ *or* $Y^{\#} = (\_, p', \_)$*, a trace* $(X\,(\mathsf{skip}, \sigma, \sigma)\,Y, X^{\#}\,(\mathsf{skip}, p', p')\,Y^{\#})$ *is also in* S*.*

Definition 22 (skip-mumbling)**.** *A set* $S \subseteq \mathsf{Traces} \times \mathsf{Traces}^{\#}$ *is closed under* skip*-mumbling, when for every trace*

$$(X\,(\alpha_1, \sigma_1, \sigma_2)\,(\alpha_2, \sigma_2, \sigma_3)\,Y, X^{\#}\,(\alpha_1, p_1, p_2)\,(\alpha_2, p_2, p_3)\,Y^{\#}) \in S$$

such that either of α_1 *or* α_2 *is* skip*, it implies that*

$$(X\,(\alpha_2 \circ \alpha_1, \sigma_1, \sigma_3)\,Y, X^{\#}\,(\alpha_2 \circ \alpha_1, p_1, p_3)\,Y^{\#}) \in S.$$

Proposition 23. $\mathcal{S}[\![\mathbb{P}]\!](p)$ *is closed under stuttering and* skip*-mumbling.*

We say that traces $(X, X^{\#})$ and $(Y, Y^{\#})$ are equal w.r.t. stuttering and skip-mumbling, written $(X, X^{\#}) =_{\dagger} (Y, Y^{\#})$, if a finite number of stutterings and skip-mumblings can be applied to both of them in order to obtain equal traces. We also let $X^{\#} =_{\dagger} Y^{\#} \triangleq \exists X, Y.\, (X, X^{\#}) =_{\dagger} (Y, Y^{\#}) \wedge X \in \gamma(X^{\#}) \wedge Y \in \gamma(Y)$.

Client-local semantics and composition. The proof structure of the Abstraction theorem requires us to give a client-local semantics to a client $\mathbb{C}$ in program $\mathbb{C}(\mathbb{L})$. For that we consider the program $\mathbb{C}(\cdot)$ which is obtained from $\mathbb{C}(\mathbb{L})$ by replacing interpretation of library's methods with $\mathbb{L}_{\mathsf{stub}} = \lambda m \in \mathsf{Methods}.\, \mathsf{skip}$. Since the command skip can always be mumbled, it is possible to say that $\mathbb{L}_{\mathsf{stub}}$ interprets bodies of methods as empty commands.

Composition. Given two local executions of $\mathbb{C}(\cdot)$ and $\mathbb{L}$, we will be interested in combining them into a safe execution of $\mathbb{C}(\mathbb{L})$. To achieve this we consider a set of interleavings defined in a standard manner as follows:

$$\varepsilon \parallel \varepsilon = \{\varepsilon\}$$
$$X\,(\alpha_1, \sigma_1, \sigma_1') \parallel \varepsilon = \varepsilon \parallel X\,(\alpha_1, \sigma_1, \sigma_1') = \{Z\,(\alpha_1, \sigma_1, \sigma_1') \mid Z \in X \parallel \varepsilon \wedge \alpha_1 \notin \mathsf{Atom_I}\}$$

$$
\begin{aligned}
X\,(\alpha_1,\sigma_1,\sigma_1') \parallel Y\,(\alpha_2,\sigma_2,\sigma_2') = \\
&\{Z\,(\alpha_1,\sigma_1,\sigma_1') \mid Z \in X \parallel Y\,(\alpha_2,\sigma_2,\sigma_2') \wedge \alpha_1,\alpha_2 \notin \mathsf{Atom_I}\} \cup \\
&\{Z\,(\alpha_2,\sigma_2,\sigma_2') \mid Z \in X\,(\alpha_1,\sigma_1,\sigma_1') \parallel Y \wedge \alpha_1,\alpha_2 \notin \mathsf{Atom_I}\} \cup \\
&\{Z\,(\alpha_1,\sigma_1,\sigma_1') \mid Z \in X \parallel Y \wedge \alpha_1 = \alpha_2 \in \mathsf{Atom_I} \wedge \sigma_1 = \sigma_1' = \sigma_2 = \sigma_2'\}
\end{aligned}
$$

In the definition above, the result of parallel composition of two traces is set of such interleavings of them, that checks if these two traces agree on interface transitions, and is undefined, if not. We enhance this definition by considering abstract traces, composition of which should be stating properties about resulting concrete traces.

Definition 24 (Safe composition)**.** *The safe composition of two pairs* $(X, X^\#)$ *and* $(Y, Y^\#)$ *of concrete and abstract traces is:*

$$
\begin{aligned}
(X, X^\#) \parallel (Y, Y^\#) \triangleq \{(Z, \overline{X}^\#, \overline{Y}^\#) \mid Z \in X \parallel Y \wedge Z \in \gamma(\overline{X}^\# \circledast \overline{Y}^\#) \wedge \\
X^\# =_\dagger \overline{X}^\# \wedge Y^\# =_\dagger \overline{Y}^\#\}
\end{aligned}
$$

Lemma 25 (Decomposition)**.** *If* $(Z, X^\#, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$*, then there are* $\overline{X}^\#, \overline{Y}^\#$ *such that:*

- $(\mathsf{client}(Z), \overline{X}^\#) \in \mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$ *and* $X^\# =_\dagger \overline{X}^\#$*;*
- $(\mathsf{lib}(Z), \overline{Y}^\#) \in \mathcal{S}[\![\mathbb{L}]\!](p_l)$ *and* $Y^\# =_\dagger \overline{Y}^\#$*; and*
- $(Z, X^\#, Y^\#) \in (\mathsf{client}(Z), \overline{X}^\#) \parallel (\mathsf{lib}(Z), \overline{Y}^\#)$.

Lemma 26 (Composition)**.** *If* $(X, X^\#) \in \mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$*,* $(Y, Y^\#) \in \mathcal{S}[\![\mathbb{L}]\!](p_l)$*, then* $(X, X^\#) \parallel (Y, Y^\#) \subseteq \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$.

Proof of Theorem 13. Let us take $(Z_1, X^\#, Y_1^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L}_1)]\!](p_c, p_l)$ and let $X = \mathsf{client}(Z_1)$ and $Y_1 = \mathsf{lib}(Z_1)$. By Lemma 25 (Decomposition), some identity transitions can be omitted from $X^\#$ and $Y_1^\#$ in order to obtain $\overline{X}^\# =_\dagger X^\#$ and $\overline{Y_1}^\# =_\dagger Y_1^\#$ with the following properties:

- $(X, \overline{X}^\#) \in \mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$, and
- $(Y_1, \overline{Y_1}^\#) \in \mathcal{S}[\![\mathbb{L}_1]\!](p_l)$, and
- $(Z_1, X^\#, Y_1^\#) \in (X, \overline{X}^\#) \parallel (Y_1, \overline{Y_1}^\#)$.

Since $\mathbb{L}_1 \sqsubseteq_{p_l} \mathbb{L}_2$ holds, there is $(Y_2, \overline{Y_2}^\#) \in \mathcal{S}[\![\mathbb{L}_2]\!](p_l)$ such that $(Y_1, \overline{Y_1}^\#) \sqsubseteq (Y_2, \overline{Y_2}^\#)$. The latter particularly means that machine states and interface transitions in Y_1 and Y_2 are the same in correspondent transitions of these two traces. Knowing that $\mathsf{lib}(Z_1) = Y_1$, we replace each library's transition of Z_1 with a correspondent transition from Y_2, obtaining a trace Z_2 as the result. Obviously, $\mathsf{client}(Z_2) = X$ and $\mathsf{lib}(Z_2) = Y_2$, and it is easy to see that $Z_2 \in (\mathsf{client}(Z_2) \parallel \mathsf{lib}(Z_2))$.

Consider $\overline{Y_1}^{\#}$ and $\overline{Y_2}^{\#}$, transitions of which are related according to the $(Y_1, \overline{Y_1}^{\#}) \sqsubseteq (Y_2, \overline{Y_2}^{\#})$ relation. By construction, $Y_1^{\#}$ can be obtained from $\overline{Y_1}^{\#}$ by a finite number of stutterings. Let us perform the same stutterings on $\overline{Y_2}^{\#}$ and obtain $Y_2^{\#}$ such that $|Y_1^{\#}| = |Y_2^{\#}|$. It is easy to see that according to such construction $X^{\#} \circledast Y_2^{\#}$ is defined, given that so does $X^{\#} \circledast Y_2^{\#}$.

We argue that $Z_2 \in \gamma(X^{\#} \circledast Y_2^{\#})$, when $Z_1 \in \gamma(X^{\#} \circledast Y_1^{\#})$. Let us take a look at each transition in $Z_1, Z_2, X^{\#} \circledast Y_1^{\#}$ and $X^{\#} \circledast Y_2^{\#}$, which are $(\alpha_1, \sigma, \sigma'), (\alpha_2, \sigma, \sigma'), (\alpha_1, p_c' * p_1', p_c' * p_1'')$ and $(\alpha_2, p_c' * p_2', p_c' * p_2'')$ correspondingly, when $\alpha_1, \alpha_2 \in \mathsf{Atom_L}$ (the case of $\alpha_1, \alpha_2 \in \mathsf{Atom_C}$ is analogous), p_c' is a client's view and the others are library's views. Since $Z_1 \in \gamma(X^{\#} \circledast Y_1^{\#})$, it is the case that $\sigma \in \lfloor p_c' * p_1' \rfloor$ and $\sigma' \in \lfloor p_c' * p_1'' \rfloor$ hold. Knowing from $Y_1^{\#} \preceq Y_2^{\#}$ that $\mathsf{skip} \Vdash \{p_1'\}\{p_2'\}$ and $\mathsf{skip} \Vdash \{p_1''\}\{p_2''\}$, which by the Frame Property of the axioms implies $\lfloor p_c' * p_1' \rfloor \subseteq \lfloor p_c' * p_2' \rfloor$ and $\lfloor p_c' * p_1'' \rfloor \subseteq \lfloor p_c' * p_2'' \rfloor$, we conclude that $\sigma \in \lfloor p_c' * p_1' \rfloor$ and $\sigma' \in \lfloor p_c' * p_1'' \rfloor$ hold as well. Consequently, $Z_2 \in \gamma(X^{\#} \circledast Y_2^{\#})$.

Overall we constructed a trace $(Z_2, X^{\#}, Y_2^{\#})$ with the following properties:

$$Z_2 \in X \parallel Y_2 \wedge Z_2 \in \gamma(X^{\#} \circledast Y_2^{\#}) \wedge \overline{X}^{\#} =_{\dagger} X^{\#} \wedge \overline{Y_2}^{\#} =_{\dagger} Y_2^{\#}$$

which means that $(Z_2, X^{\#}, Y_2^{\#}) \in (X, \overline{X}^{\#}) \parallel (Y_2, \overline{Y_2}^{\#})$. We apply Lemma 26 (Composition) to traces $(X, \overline{X}^{\#})$ and $(Y_2, \overline{Y_2}^{\#})$, according to which the following holds:

$$(Z_2, X^{\#}, Y_2^{\#}) \in (X, \overline{X}^{\#}) \parallel (Y_2, \overline{Y_2}^{\#}) \subseteq \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L}_2)]\!](p_c, p_l)$$

Thus, we have shown that for any $(Z_1, X_1^{\#}, Y_1^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L}_1)]\!](p_c, p_l)$ we are able to find $(Z_2, X_2^{\#}, Y_2^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L}_2)]\!](p_c, p_l)$ so that

$$(X_1, X_1^{\#}) = (X_2, X_2^{\#}), \mathsf{views}(Y_1^{\#}) \preceq \mathsf{views}(Y_2^{\#}) \text{ and } \mathsf{states}(Y_1) = \mathsf{states}(Y_2),$$

which concludes the proof. □

Chapter 6

Auxillary proofs for the Abstraction theorem

6.1 Proof of the Decomposition Lemma

Lemma 25 (Decomposition)**.** *If* $(Z, X^\#, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$*, then there are* $\overline{X}^\#, \overline{Y}^\#$ *such that:*

- $(\mathsf{client}(Z), \overline{X}^\#) \in \mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$ *and* $X^\# =_\dagger \overline{X}^\#$*;*
- $(\mathsf{lib}(Z), \overline{Y}^\#) \in \mathcal{S}[\![\mathbb{L}]\!](p_l)$ *and* $Y^\# =_\dagger \overline{Y}^\#$*; and*
- $(Z, X^\#, Y^\#) \in (\mathsf{client}(Z), \overline{X}^\#) \parallel (\mathsf{lib}(Z), \overline{Y}^\#)$.

Proof. Consider an arbitrary $(Z, X^\#, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$. We explain the construction of an abstract trace $\overline{X}^\# =_\dagger X^\#$ such that $(\mathsf{client}(Z), \overline{X}^\#) \in \mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$, and the construction of $\overline{Y}^\#$ is analogous.

By Definition 11 of internally safe semantics $\mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$, $Z \in \gamma(X^\# \circledast Y^\#)$, $\mathsf{valid}(X^\#)$ and $\mathsf{valid}(Y^\#)$ hold. When $Z \in \gamma(X^\# \circledast Y^\#)$, for each client's transition $(\alpha, \sigma, \sigma')$ in Z there is a corresponding (α, p_c', p_c'') in $X^\#$, and for each library's transition in Z there is a corresponding $(\mathsf{skip}, p_c''', p_c''')$. Let us consider $\overline{X}^\#$, which is obtained by omitting transitions from $X^\#$ that are corresponding to library transitions. It is easy to see that $\mathsf{valid}(\overline{X}^\#)$ holds whenever $\mathsf{valid}(X^\#)$ does.

We further argue that $Z \in \gamma(X^\# \circledast Y^\#)$ implies that $\mathsf{client}(Z) \in \gamma(\overline{X}^\#)$. Consider every transition $(\alpha, \sigma, \sigma')$ in $\mathsf{client}(Z)$. Since the latter projection is obtained by omitting library transitions from Z and $Z \in \gamma(X^\# \circledast Y^\#)$, there is a transition $(\alpha, p_c' * p_l', p_c'' * p_l')$ in $X^\# \circledast Y^\#$ such that $\sigma \in \lfloor p_c' * p_l' \rfloor$ and $\sigma' \in \lfloor p_c'' * p_l' \rfloor$. By Property 4, $\sigma \in \lfloor p_c' \rfloor$ and $\sigma' \in \lfloor p_c'' \rfloor$. Consequently, $\mathsf{client}(Z) \in \gamma(\overline{X}^\#)$ holds. Together with $\mathsf{valid}(\overline{X}^\#)$ this gives us that $(\mathsf{client}(Z), \overline{X}^\#) \in \mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$.

It remains to show that $(Z, X^\#, Y^\#) \in (\mathsf{client}(Z), \overline{X}^\#) \parallel (\mathsf{lib}(Z), \overline{Y}^\#)$. However, it is easy to see that $Z \in \mathsf{client}(Z) \parallel \mathsf{lib}(Z)$. Knowing that $Z \in \gamma(X^\# \circledast Y^\#)$, we

conclude that safe composition $(\mathsf{client}(Z), \overline{X}^{\#}) \parallel (\mathsf{lib}(Z), \overline{Y}^{\#}$ contains $(Z, X^{\#}, Y^{\#})$ by Definition 24. □

6.2 Proof of the Composition Lemma

Lemma 26 (Composition)**.** *If* $(X, X^{\#}) \in \mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$*,* $(Y, Y^{\#}) \in \mathcal{S}[\![\mathbb{L}]\!](p_l)$*, then* $(X, X^{\#}) \parallel (Y, Y^{\#}) \subseteq \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$.

Proof. Every trace $(Z, \overline{X}^{\#}, \overline{Y}^{\#}) \in (X, X^{\#}) \parallel (Y, Y^{\#})$ in order to be in $\mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$ has to satisfy the following:

- $Z \in \mathcal{T}[\![\mathbb{C}(\mathbb{L})]\!]$;
- $Z \in \gamma(\overline{X}^{\#} \circledast \overline{Y}^{\#})$;
- $\mathsf{valid}(\overline{X}^{\#})$ and $\mathsf{valid}(\overline{Y}^{\#})$.

It is easy to see that $Z \in (X, X^{\#}) \parallel (Y, Y^{\#}) \subseteq X \parallel Y \subseteq \mathcal{T}[\![\mathbb{C}(\mathbb{L})]\!]$, from which the first requirement follows. Also, by Definition 24 of safe composition, $Z \in \gamma(\overline{X}^{\#} \circledast \overline{Y}^{\#})$ holds as well. It remains to show that abstract traces $\overline{X}^{\#}$ and $\overline{Y}^{\#}$ are valid.

By the premise of the lemma, $(X, X^{\#}) \in \mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$. Knowing that $\overline{X}^{\#} =_{\dagger} X^{\#}$ and that $\mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$ is closed under stuttering and skip-mumbling, we conclude that $(\_, \overline{X}^{\#}) \in \mathcal{S}[\![\mathbb{C}(\cdot)]\!](p_c)$. Consequently, $\mathsf{valid}(\overline{X}^{\#})$ holds, and analogously does $\mathsf{valid}(\overline{Y}^{\#})$. □

6.3 Proof of Corollary 18

The statement of Corollary 18 requires a strong property from the best axioms, which tells that $\forall r.\, q * r \in \mathsf{SP}(\alpha, p * r)$ must hold whenever $q \in \mathsf{SP}(\alpha, p)$ holds. Consider $q \in \mathsf{SP}(\alpha, p)$ and let view q' be such that $\alpha \Vdash \{p\}\{q'\}$ holds; then $\lfloor q \rfloor \subseteq \lfloor q' \rfloor$ holds. Let us take any $r \in \mathsf{View}$. By the Frame Property of axioms, $\alpha \Vdash \{p * r\}\{q' * r\}$ holds. Since $q * r \in \mathsf{SP}(\alpha, p * r)$, then $\lfloor q * r \rfloor \subseteq \lfloor q' * r \rfloor$. Overall we obtained that if $q \in \mathsf{SP}(\alpha, p)$ and $\alpha \Vdash \{p\}\{q'\}$, then $\forall r.\, \lfloor q * r \rfloor q' * r$. Particularly, if $q' \in \mathsf{SP}(\alpha, p)$ as well, then $\forall r.\, \lfloor q * r \rfloor = \lfloor q' * r \rfloor$. We use this observation in further proofs.

Proposition 27. *When best axioms of the View Framework are closed under Frame property and are generated from semantics, if* $\forall r.\, \lfloor p_1 * r \rfloor = \lfloor p_2 * r \rfloor$*, then* $\mathsf{safe}(\mathbb{P}, p_1) \iff \mathsf{safe}(\mathbb{P}, p_2)$.

Proof. We prove that $\mathsf{safe}(\mathbb{P}, p_1) \implies \mathsf{safe}(\mathbb{P}, p_2)$, and the proof in the other direction is analogous. Let us assume that $\mathsf{safe}(\mathbb{P}, p_1)$ holds, i.e.:

$$\forall \alpha, \mathbb{P}', \sigma, \sigma'.\, \sigma \in \lfloor p_1 \rfloor \wedge \langle \mathbb{P}, \sigma \rangle \overset{\alpha}{\Longrightarrow} \langle \mathbb{P}', \sigma' \rangle \implies \exists p'.\, p' \in \mathsf{SP}(\alpha, p_1) \wedge \mathsf{safe}(\mathbb{P}', p').$$

Consider any transition $\langle \mathbb{P}, \sigma \rangle \xRightarrow{\alpha} \langle \mathbb{P}', \sigma' \rangle$ so that $\sigma \in \lfloor p_1 \rfloor$. Then there exists $p' \in \mathsf{SP}(\alpha, p_1)$ and $\mathsf{safe}(\mathbb{P}', p')$ holds. We further demonstrate that $p' \in \mathsf{SP}(\alpha, p_2)$ holds. The following conditions should be met for that:

- $\alpha \Vdash \{p_2\}\{p'\}$; and
- $\forall q.\, \alpha \Vdash \{p_2\}\{q\} \implies \lfloor p' \rfloor \subseteq \lfloor q \rfloor$.

Note that since $\forall r.\, f_\alpha(\lfloor p_2 * r \rfloor) = f_\alpha(\lfloor p_1 * r \rfloor) \subseteq \lfloor p' * r \rfloor$ holds, the axioms $\alpha \Vdash \{p_2\}\{p'\}$ is generated from semantics, so the first condition is satisfied. Since $p' \in \mathsf{SP}(\alpha, p_1)$:

$$\forall q.\, \alpha \Vdash \{p_1\}\{q\} \implies \lfloor p' \rfloor \subseteq \lfloor q \rfloor$$

It is easy to see that analogously to p', for every view q, $\alpha \Vdash \{p_1\}\{q\}$ if and only if $\alpha \Vdash \{p_2\}\{q\}$. Consequently, the second condition for $p' \in \mathsf{SP}(\alpha, p_2)$ holds as well.

Overall we have shown for any transition $\langle \mathbb{P}, \sigma \rangle \xRightarrow{\alpha} \langle \mathbb{P}', \sigma' \rangle$ so that $\sigma \in \lfloor p_1 \rfloor = \lfloor p_2 \rfloor$, there exists $p' \in \mathsf{SP}(\alpha, p_1)$ and $\mathsf{safe}(\mathbb{P}', p')$ holds. Consequently, by Definition 5, $\mathsf{safe}(\mathbb{P}, p_2)$ holds. □

Lemma 28. *If best axioms of the View Framework are closed under Frame property and are generated from semantics, then:*

$$\mathsf{safe}(\mathbb{P}, p) \wedge ((\alpha, \sigma, \sigma')\, Z, (\alpha, p, p')\, Z^\#) \in \mathcal{S}[\![\mathbb{P}]\!](p) \implies \exists \mathbb{P}'.\, \mathsf{safe}(\mathbb{P}', p') \wedge (Z, Z^\#) \in \mathcal{S}[\![\mathbb{P}']\!](p').$$

Proof. Let us take any trace $((\alpha, \sigma, \sigma')\, Z, (\alpha, p, p')\, Z^\#) \in \mathcal{S}[\![\mathbb{P}]\!](p)$. By Definition 6 of safe semantics, there is a transition $\langle \mathbb{P}, \sigma \rangle \xRightarrow{\alpha} \langle \mathbb{P}', \sigma' \rangle$, and also $p' \in \mathsf{SP}(\alpha, p)$ $\sigma \in \lfloor p \rfloor$ both hold. Let us unfold $\mathsf{safe}(\mathbb{P}, p)$ by Definition 5:

$$\forall \alpha, \mathbb{P}', \sigma, \sigma'.\, \sigma \in \lfloor p \rfloor \wedge \langle \mathbb{P}, \sigma \rangle \xRightarrow{\alpha} \langle \mathbb{P}', \sigma' \rangle \implies \exists p''.\, p'' \in \mathsf{SP}(\alpha, p) \wedge \mathsf{safe}(\mathbb{P}', p'').$$

The implication above holds of all possible transitions, so it must hold of $\langle \mathbb{P}, \sigma \rangle \xRightarrow{\alpha} \langle \mathbb{P}', \sigma' \rangle$, since $\sigma \in \lfloor p \rfloor$. Consequently, we obtain that $\exists p''.\, p'' \in \mathsf{SP}(\alpha, p) \wedge \mathsf{safe}(\mathbb{P}', p'')$.

Both p' and p'' are strongest postconditions from $\mathsf{SP}(\alpha, p)$, so $\forall r \in \mathsf{View}.\, \lfloor p' * r \rfloor = \lfloor p'' * r \rfloor$ necessarily holds. By Proposition 27, $\mathsf{safe}(\mathbb{P}', p')$ holds then. □

Analogously, we formulate the same result for internal safety.

Lemma 29. *If best axioms of the View Framework are closed under Frame property and are generated from semantics, then:*

$$\mathsf{safeint}(\mathbb{P}, p_c, p_l) \wedge ((\alpha, \sigma, \sigma')\, Z, (\alpha, p_c, p_c')\, X^\#, (\mathsf{skip}, p_l, p_l)\, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{P}]\!](p_c, p_l) \implies \exists \mathbb{P}'.\, \mathsf{safeint}(\mathbb{P}', p_c', p_l) \wedge (Z, X^\#, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{P}']\!](p_c', p_l), \textit{ and}$$

$$\mathsf{safeint}(\mathbb{P}, p_c, p_l) \wedge ((\alpha, \sigma, \sigma')\, Z, (\mathsf{skip}, p_c, p_c)\, X^\#, (\alpha, p_l, p_l')\, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{P}]\!](p_c, p_l) \implies \exists \mathbb{P}'.\, \mathsf{safeint}(\mathbb{P}', p_c, p_l') \wedge (Z, X^\#, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{P}']\!](p_c, p_l').$$

Proposition 30.

$$\forall Z, Z^{\#}, X^{\#}, Y^{\#}.(Z, Z^{\#}) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c{*}p_l){\wedge}(Z, X^{\#}, Y^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l) \implies \mathsf{views}(Z^{\#}) \preceq \mathsf{views}(X^{\#} \circledast Y^{\#})$$

Proof. The case of $Z = \varepsilon$ is trivial. Let us prove the statement by induction on the length n of Z_n. When $n = 1$, $Z_1^{\#} = (\alpha, p_c * p_l, p')$ and $X_1^{\#} \circledast Y_1^{\#} = (\alpha, p_c * p_l, p'_c * p'_l)$. According to definitions of safe and internally safe semantics, $p' \in \mathsf{SP}\alpha, p_c * p_l$ and $\alpha \Vdash \{p_c * p_l\}\{p'_c * p'_l\}$. Since p' is strongest postcondition, $p' \preceq p'_c * p'_l$.

Now let us show induction step. Consider $Z_{n+1}^{\#} = Z_n^{\#}(\alpha, p', p'')$ and $X_{n+1}^{\#} \circledast Y_{n+1}^{\#} = X_n^{\#} \circledast Y_n^{\#}\,(\alpha, p'_c * p'_l, p''_c * p''_l)$. By induction hypothesis, $p' \preceq p'_c * p'_l$. According to definitions of safe and internally safe semantics, $p'' \in \mathsf{SP}\alpha, p'$ and $\alpha \Vdash \{p'_c{*}p'_l\}\{p''_c{*}p''_l\}$. Note that $p' \preceq p'_c{*}p'_l$ and $\alpha \Vdash \{p'_c{*}p'_l\}\{p''_c{*}p''_l\}$ together imply $\alpha \Vdash \{p'\}\{p''_c{*}p''_l\}$. Since p'' is a strongest postcondition for α and p', $p'' \preceq p''_c * p''_l$ holds, which concludes the proof by induction. □

Lemma 16. *When best axioms of the View Framework are closed under Frame property and are generated from semantics, given* $\mathsf{safeint}(\mathbb{C}(\mathbb{L}), p_c, p_l)$*, any abstract trace* $Z^{\#}$ *of a trace* Z *can be overapproximated by local abstract traces* $X^{\#}$ *and* $Y^{\#}$*:*

$$\mathsf{safeint}(\mathbb{C}(\mathbb{L}), p_c, p_l) \implies \forall Z, Z^{\#}.\,(Z, Z^{\#}) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l) \implies \exists X^{\#}, Y^{\#}.\,(Z, X^{\#}, Y^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l) \wedge \mathsf{views}(Z^{\#}) \preceq \mathsf{views}(X^{\#} \circledast Y^{\#})$$

Proof. We prove the following statement by induction on n, which is the length of traces $Z_n, Z_n^{\#}, X_n^{\#}, Y_n^{\#}$:

$$\Phi(n) = \mathsf{safeint}(\mathbb{C}(\mathbb{L}), p_c, p_l) \implies \forall Z_n, Z_n^{\#}.\,(Z_n, Z_n^{\#}) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l) \implies \exists X_n^{\#}, Y_n^{\#}.\,(Z_n, X_n^{\#}, Y_n^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l) \wedge \mathsf{views}(Z^{\#}) \preceq \mathsf{views}(X^{\#} \circledast Y^{\#})).$$

In the base case of $n = 0$, $\Phi(0)$ holds trivially: the statement about empty traces follows straightforwardly. Let us prove the induction step $\Phi(n) \implies \Phi(n+1)$. Consider a trace $(Z_{n+1}, Z_{n+1}^{\#}) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l)$ and let $Z_{n+1} = Z_n\,(\alpha, \sigma, \sigma')$ and $Z_{n+1}^{\#} = Z_n^{\#}\,(\alpha, p', p'')$. By $\Phi(n)$, there is $(Z_n, X_n^{\#}, Y_n^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$.

Let $\mathbb{P}$ be a program, which is obtained by performing transitions of Z_n. By applying Lemma 28 n times to $(Z_n, X_n^{\#}, Y_n^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$, we obtain that $\mathsf{safeint}(\mathbb{P}, p'_c, p'_l)$ holds, which means the following:

$$\forall \alpha, \mathbb{P}', \sigma, \sigma'.\,\sigma \in \lfloor p'_c * p'_l \rfloor \wedge \langle \mathbb{P}, \sigma \rangle \xrightarrow{\alpha} \langle \mathbb{P}', \sigma' \rangle \implies (\alpha \in \mathsf{Atom_C} \implies \exists p''_c.\,p''_c \in \mathsf{SP}(\alpha, p'_c) \wedge \mathsf{safeint}(\mathbb{P}', p''_c, p'_l)) \wedge (\alpha \in \mathsf{Atom_L} \implies \exists p''_l.\,p''_l \in \mathsf{SP}(\alpha, p'_l) \wedge \mathsf{safeint}(\mathbb{P}', p'_c, p''_l)). \quad (6.1)$$

A trace $Z_n\,(\alpha, \sigma, \sigma')$ from safe semantics is a transition trace, which according to Definition 1 means that there is a transition $\langle \mathbb{P}, \sigma \rangle \overset{\alpha}{\Longrightarrow} \langle \mathbb{P}', \sigma' \rangle$. By Proposition 30, $\mathsf{views}(Z_n^{\#}) \preceq \mathsf{views}(X_n^{\#} \circledast Y_n^{\#})$ holds, and since p', p'_c, p'_l are the last

views in $Z_n^\#, X_n^\#, Y_n^\#$ correspondingly, $p' \preceq p'_c * p'_l$. Since Z_{n+1} in safe semantics, $Z_n(\alpha, \sigma, \sigma') \in \gamma(Z_n^\#(\alpha, p', p''))$, so $\sigma \in \lfloor p' \rfloor \subseteq p'_c * p'_l$. Altogether all these facts give us that the premise of $\mathsf{safeint}(\mathbb{P}, p'_c, p'_l)$ holds of transition $\langle \mathbb{P}, \sigma \rangle \xRightarrow{\alpha} \langle \mathbb{P}', \sigma' \rangle$. Consequently:

$$(\alpha \in \mathsf{Atom_C} \implies \exists p''_c.\, p''_c \in \mathsf{SP}(\alpha, p'_c)) \wedge (\alpha \in \mathsf{Atom_L} \implies \exists p''_l.\, p''_l \in \mathsf{SP}(\alpha, p'_l))$$

It is easy to see that in cases of α performed by a client and a library correspondingly, $(Z_{n+1}, X_n^\#(\alpha, p'_c, p''_c), Y_n^\#(\mathsf{skip}, p'_l, p'_l)) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$ and $(Z_{n+1}, X_n^\#(\mathsf{skip}, p'_c, p'_c), Y_n^\#(\alpha, p'_l, p''_l)) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$ holds. □

Lemma 17. *When best axioms of the View Framework are closed under Frame property,*

$$\forall Z, X^\#, Y^\#.\, (Z, X^\#, Y^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l) \implies (Z, X^\# \circledast Y^\#) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l).$$

Proof. By induction on the length of traces.

$$\Phi(n) = (\forall p, q, \alpha.\, q \in \mathsf{SP}(\alpha, p) \implies q * r \in \mathsf{SP}(\alpha, p * r)) \implies \forall Z_n, X_n^\#, Y_n^\#.\, (Z_n, X_n^\#, Y_n^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l) \implies (Z, X^\# \circledast Y^\#) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l)$$

Consider any traces $Z_{n+1}, X_{n+1}^\#, Y_{n+1}^\#$ of the length $n+1$ such that $(Z_{n+1}, X_{n+1}^\#, Y_{n+1}^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$. By induction hypothesis, if its prefix is $(Z_n, X_n^\#, Y_n^\#)$, then $(Z_n, X_n^\# \circledast Y_n^\#) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l)$. Let us consider the last transitions $(\alpha, \sigma, \sigma')$, (α, p'_c, p''_c) and $(\mathsf{skip}, p'_l, p'_l)$ in $Z_{n+1}, X_{n+1}^\#$ and $Y_{n+1}^\#$ respectively (which is the case of the client's transition, and the other case is analogous). Since Z_{n+1} is internally safe, $\sigma \in \lfloor p'_c * p'_l \rfloor$ and $p''_c \in \mathsf{SP}(\alpha, p'_c)$, so $\alpha \Vdash \{p'_c * p'_l\}\{p''_c * p'_l\}$ holds. Moreover, by Property 14, $p''_c * p'_l \in \mathsf{SP}(\alpha, p'_c * p'_l)$, which is sufficient for $(Z_{n+1}, X_{n+1}^\# \circledast Y_{n+1}^\#) \in \mathcal{S}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l)$ to hold. □

6.4 Proof of Corollary 20

Corollary 20. *If a client $\mathbb{C}$ and libraries $\mathbb{L}_1$ and $\mathbb{L}_2$ are such that $\mathsf{safeint}(\mathbb{C}(\mathbb{L}_1), p_c, p_l)$ and $\mathbb{L}_1 \sqsubseteq_{p_l} \mathbb{L}_2$ hold, then:*

$$\forall Z_1 \in \mathcal{S}_{\mathsf{closed}}[\![\mathbb{C}(\mathbb{L}_1)]\!](p_c * p_l).\, \exists Z_2 \in \mathcal{S}_{\mathsf{closed}}[\![\mathbb{C}(\mathbb{L}_2)]\!](p_c * p_l).\, \mathsf{client}(Z_1) = \mathsf{client}(Z_2) \wedge \mathsf{states}(\mathsf{lib}(Z_1)) = \mathsf{states}(\mathsf{lib}(Z_2)).$$

Proof. Consider any $Z_1 \in \mathcal{S}_{\mathsf{closed}}[\![\mathbb{C}(\mathbb{L}_1)]\!](p_c * p_l)$. We first show that $\exists X_1^\#, Y_1^\#.\, (Z_1, X_1^\#, Y_1^\#) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L})]\!](p_c, p_l)$. Specifically, we construct $X_1^\#$ and $Y_1^\#$ that satisfy the following:

- $Z_1 \in \gamma(X_1^\# \circ Y_1^\#)$, and
- $\mathsf{valid}(X_1^\#)$ and $\mathsf{valid}(Y_1^\#)$ hold.

According to Definition 11 of internal safety:

$$\forall \alpha, \mathbb{P}, \sigma, \sigma'.\, \sigma \in \lfloor p_c * p_l \rfloor \wedge \langle \mathbb{C}(\mathbb{L}), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \mathbb{P}, \sigma' \rangle \implies (\alpha \in \mathsf{Atom_C} \implies \exists p'_c.\, p'_c \in \mathsf{SP}(\alpha, p_c) \wedge \mathsf{safeint}(\mathbb{P}, p'_c, p_l)) \wedge (\alpha \in \mathsf{Atom_L} \implies \exists p'_l.\, p'_l \in \mathsf{SP}(\alpha, p_l) \wedge \mathsf{safeint}(\mathbb{P}, p_c, p'_l)). \quad (6.2)$$

When $Z_1 = \varepsilon$, the statement of the theorem is trivial. Let $Z_1 = \{(\alpha_i, \sigma_i, \sigma_{i+1})\}_{i=1}^{k}$, where $k \geq 1$. Since $Z_1 \in \mathcal{S}_{\mathsf{closed}}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l)$, $\sigma_1 \in \lfloor p_c * p_l \rfloor$ and $\exists \mathbb{P}.\, \langle \mathbb{C}(\mathbb{L}), \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle \mathbb{P}, \sigma' \rangle$. The premise of (6.2) must hold of this transition, so:

$$(\alpha \in \mathsf{Atom_C} \implies \exists p'_c.\, p'_c \in \mathsf{SP}(\alpha, p_c) \wedge \mathsf{safeint}(\mathbb{P}, p'_c, p_l)) \wedge (\alpha \in \mathsf{Atom_L} \implies \exists p'_l.\, p'_l \in \mathsf{SP}(\alpha, p_l) \wedge \mathsf{safeint}(\mathbb{P}, p_c, p'_l)). \quad (6.3)$$

Consider three cases: *(a)* $\alpha \in \mathsf{Atom_C} \setminus \mathsf{Atom_I}$, *(b)* $\alpha \in \mathsf{Atom_L} \setminus \mathsf{Atom_I}$ and *(c)* $\alpha \in \mathsf{Atom_I}$. In the case *(a)*, $\exists p'_c.\, p'_c \in \mathsf{SP}(\alpha, p_c) \wedge \mathsf{safeint}(\mathbb{P}, p'_c, p_l)$. We let the first transition of $X_1^{\#}$ and $Y_1^{\#}$ be (α, p_c, p'_c) and $(\mathsf{skip}, p_l, p_l)$. Note that $\sigma_2 \in \lfloor p'_c * p_l \rfloor$, because $\alpha \Vdash \{p_c * p_l\}\{p'_c * p_l\}$ holds by the Frame Property of the axiom and $\sigma_2 \in f_\alpha(\sigma_1) \subseteq f_\alpha(\lfloor p_c * p_l \rfloor) \subseteq \lfloor p'_c * p_l \rfloor$. In the case *(b)*, $\exists p'_l.\, p'_l \in \mathsf{SP}(\alpha, p_l) \wedge \mathsf{safeint}(\mathbb{P}, p_c, p'_l)$, and we let the first transition of $X_1^{\#}$ and $Y_1^{\#}$ be $(\mathsf{skip}, p_c, p_c)$ and (α, p_l, p'_l). Also, it is easy to show that $\sigma_2 \in \lfloor p_c * p'_l \rfloor$. Finally, in the case *(c)* we let the first transition of $X_1^{\#}$ and $Y_1^{\#}$ be (α, p_c, p_c) and (α, p_l, p_l). Moreover, $\sigma_2 \in \lfloor p_c * p_l \rfloor$ holds. By continuing construction until the end of Z_1, we obtain $X_1^{\#}$ and $Y_1^{\#}$ satisfying the requirements of internally safe semantics.

According to Theorem 13, the following holds:

$$\forall (Z_1, X_1^{\#}, Y_1^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L}_1)]\!](p_c, p_l).\, \exists (Z_2, X_2^{\#}, Y_2^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L}_2)]\!](p_c, p_l).\; Z_1^{\#} \preceq X_2^{\#} \circledast Y_2^{\#} \wedge \mathsf{client}(Z_1) = \mathsf{client}(Z_2) \wedge \mathsf{states}(\mathsf{lib}(Z_1)) = \mathsf{states}(\mathsf{lib}(Z_2)).$$

Let us take $(Z_2, X_2^{\#}, Y_2^{\#}) \in \mathcal{S}_{\mathsf{int}}[\![\mathbb{C}(\mathbb{L}_2)]\!](p_c, p_l)$ from the formula above. Note that $\mathsf{client}(Z_1) = \mathsf{client}(Z_2)$ and $\mathsf{states}(\mathsf{lib}(Z_1)) = \mathsf{states}(\mathsf{lib}(Z_2))$, which means that concrete states are not changed by the environment in Z_2 as well as in Z_1. Consequently, $Z_2 \in \mathcal{S}_{\mathsf{closed}}[\![\mathbb{C}(\mathbb{L})]\!](p_c * p_l)$. □

6.5 Proof of Proposition 8

Proposition 8. *If* $\mathsf{safe}(C, p)$ *holds, then safe semantics may be represented as a set of pairs* $(Z, Z^{\#})$ *of valid abstract traces* $Z^{\#}$ *and their concretisations* Z*:*

$$\mathcal{S}[\![\mathbb{C}(\cdot)]\!](p) = \{(Z, Z^{\#}) \mid Z \in \mathcal{T}[\![\mathbb{C}(\cdot)]\!] \wedge Z^{\#} = (\_, p, \_)_{\_} \in \mathsf{Traces}^{\#} \wedge Z \in \gamma(Z^{\#}) \wedge \mathsf{valid}(Z^{\#})\}.$$

Proof. Let $S : \mathsf{Prog} \times \mathsf{View} \to \mathsf{Traces} \times \mathsf{Traces}^{\#}$ be such a function that $S(\mathbb{P}, p)$ would be a set of traces $(Z, Z^{\#})$ satisfying the property:

$$Z \in \mathcal{T}[\![\mathbb{C}(\cdot)]\!] \wedge Z^{\#} = (\_, p, \_)_{\_} \in \mathsf{Traces}^{\#} \wedge Z \in \gamma(Z^{\#}) \wedge \mathsf{valid}(Z^{\#}). \quad (6.4)$$

We first prove that $\forall \mathbb{P}, p.\, \mathcal{S}[\![\mathbb{P}]\!](p) \subseteq S(\mathbb{P}, p)$, and then do a proof in the opposite direction. Consider any trace from $\mathcal{S}[\![\mathbb{C}(\cdot)]\!](p)$. When a trace is empty, the statement holds trivially, so we further let it be $((\alpha_1, \sigma_1, \sigma_1')\, Z, (\alpha_1, p_1, p_2)\, Z^{\#})$. By unfolding Definition 6, we get the following property of it (letting $\mathbb{P}_1 = \mathbb{P}$ and $p_1 = p$):

$$\exists \mathbb{P}_2.\, \sigma_1 \in \lfloor p_1 \rfloor \wedge \langle \mathbb{P}_1, \sigma_1 \rangle \xRightarrow{\alpha_1} \langle \mathbb{P}_2, \sigma_1' \rangle \wedge p_2 \in \mathsf{SP}(\alpha_1, p_1) \wedge (Z, Z^{\#}) \in \mathcal{S}[\![\mathbb{P}_2]\!](p_2)$$

Additionally, from $p_2 \in \mathsf{SP}(\alpha_1, p_1)$ it can be deduced that $\alpha_1(\lfloor p_1 \rfloor) \subseteq \lfloor p_2 \rfloor$. Given that $\sigma_1' \in \alpha_1(\sigma_1)$ and $\sigma_1 \in \lfloor p_1 \rfloor$, we conclude that $\sigma_1' \in \lfloor p_2 \rfloor$. Since $(Z, Z^{\#})$ is a finite trace of length $|Z| = n$, by unfolding further by definition, the following property of the whole trace $(\{(\alpha_i, \sigma_i, \sigma_i'), (\alpha_i, p_i, p_{i+1})\}_{i=1}^{n})$ is obtained:

$$\exists \mathbb{P}_2, \ldots, \mathbb{P}_n.\, \forall i.\, 1 \leq i \leq n \implies \sigma_i \in \lfloor p_i \rfloor \wedge \sigma_i' \in \lfloor p_{i+1} \rfloor \wedge \langle \mathbb{P}_i, \sigma_i \rangle \xRightarrow{\alpha_i} \langle \mathbb{P}_{i+1}, \sigma_i' \rangle \wedge p_{i+1} \in \mathsf{SP}(\alpha_i, p_i)$$

It is easy to see that $\{(\alpha_i, \sigma_i, \sigma_i') \in \gamma((\alpha_i, p_i, p_{i+1})\}_{i=1}^{n})$, $\{(\alpha_i, \sigma_i, \sigma_i') \in \mathcal{T}[\![\mathbb{P}]\!]$ and $\mathsf{valid}((\alpha_i, p_i, p_{i+1})\}_{i=1}^{n})$ all hold, which concludes the first part of the proof.

Now we show that $S(\mathbb{P}, p) \in \mathcal{S}[\![\mathbb{P}]\!](p)$ holds. Consider a trace $Z = \{(\alpha_i, \sigma_i, \sigma_i')\}_{i=1}^{n}$ and an abstract trace $Z_n^{\#} = \{(\alpha_i, p_i, p_{i+1})\}_{i=1}^{n}$ such that $(Z, Z^{\#}) \in S(\mathbb{P}, p)$. Since Z is a transition trace, there is a sequence of programs $\mathbb{P}_1 = \mathbb{P}, \mathbb{P}_2, \ldots, \mathbb{P}_{n+1}$ and transitions $\langle \mathbb{P}_i, \sigma_i \rangle \xRightarrow{\alpha_i} \langle \mathbb{P}_{i+1}, \sigma_i' \rangle$ $(1 \leq i \leq n)$.

Let $(Z_0, Z_0^{\#}) = (\varepsilon, \varepsilon^{\#})$. It is easy to see that $(Z_0, Z_0^{\#}) \in \mathcal{S}[\![\mathbb{P}_{n+1}]\!](p_{n+1})$. Now consider $(Z_k, Z_k^{\#}) = ((\alpha_k, \sigma_k, \sigma_k')\, Z_{k-1}, (\alpha_k, p_k, p_{k+1})\, Z_{k-1}^{\#})$, assuming that $(Z_{k-1}, Z_{k-1}^{\#}) \in \mathcal{S}[\![\mathbb{P}_k]\!](p_k)$. Since $Z \in \gamma(Z^{\#})$ and $\mathsf{valid}(Z^{\#})$, necessarily $\sigma_k \in \lfloor p_k \rfloor$ and $p_{k+1} \in \mathsf{SP}(\alpha_k, p_k)$. We obtained that:

$$\sigma_k \in \lfloor p_k \rfloor \wedge \langle \mathbb{P}_k, \sigma_k \rangle \xRightarrow{\alpha_k} \langle \mathbb{P}_{k+1}, \sigma_k' \rangle \wedge p_{k+1} \in \mathsf{SP}(\alpha_k, p_k) \wedge (Z_{k-1}, Z_{k-1}^{\#}) \in \mathcal{S}[\![\mathbb{P}_{k+1}]\!](p_{k+1}),$$

which implies that $(Z_k, Z_k^{\#}) \in \mathcal{S}[\![\mathbb{P}_k]\!](p_k)$ by definition. By constructing $Z_0, Z_1, \ldots, Z_n$ as described, we obtain that $(Z, Z^{\#}) = (Z_n, Z_n^{\#}) \in \mathcal{S}[\![\mathbb{P}_1]\!](p_1) = \mathcal{S}[\![\mathbb{P}]\!](p)$. □

Chapter 7

Conclusions and future work

In this work we suggested an approach to the library abstraction without assuming any information hiding in the components. We defined the library abstraction relation that enables modular proofs about programs consisting of two components, the client and the library, communicating via shared memory w.r.t. a certain contract. We formalised the contract in the definition of the internal safety and internally safe semantics, and in the Abstraction Theorem we demonstrated that internally safe traces of the client using the library are can be reconstructed by the client using an abstracted version of the library. This fact means that in a proof about the client it is sound to replace a library with its specification.

Our work lays the foundation for future correctness proofs for implementations of concurrent algorithms interacting on the shared memory. We intend to apply our approach to examples such as Michael and Scott's non-blocking queue algorithm illustrated in Chapter 2 as the next step of development of this work. To be able to deal with a wider class of examples, we also would like to extend our definitions to deal with partial information hiding between components, when only a part of the address space is being shared.

We also hope that it should be possible to develop a logic for establishing the proposed notion of linearizability formally, based on existing logics for proving safety properties generalised by the View Framework.

Bibliography

[1] Stephen Brookes. Full abstraction for a shared variable parallel language. In *LICS*, pages 98–109, 1993.

[2] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In *POPL*, 2013.

[3] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* 2008.

[4] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 1990.

[5] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[6] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[7] Peter O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.

[8] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.