Contents

1. CRIStAL laboratry

1.1 History

CRIStAL[1] (Research Center in Computer Science, Signal and Automatic Control of Lille), founded on the 1st of January 2015, is a laboratory of CNRS[2] (National Center for Scientific Research), Lille 1 university and Centrale Lille in partnership with Lille 3 University , Inria (French National Institute for computer science and applied mathematics) and Mines Telecom Institute. It is the result of the fusion of LAGIS[4] (Laboratory of Automatic Control, Computer Engineering and Signal) and LIFL[4] (Laboratory of Fundamental Computing of Lille) to federate their complementary competencies in information sciences. It is a member the interdisciplinary research institute IRCICA[5] (Research Institute on Software and Hardware Components for Advanced Information and Communication in Lille).

CRIStAL is located in Villeneuve d'Ascq city in Lille, the capital of the Hauts-de-France region and the prefecture of the Nord department and the fourth largest urban area in France after Paris, Lyon and Marseille.Chaired by *Prof.Olivier Colot*, it harbors about 430 personnel with exactly 228 permanents and more than 200 non-permanents. Permanent researchers are divided among more than 30 teams working on different themes and projects.

Figure 1.1: CRIStAL laboratry Location

[1] "Centre de Recherche en Informatique, Signal et Automatique de Lille"
[2] "Centre national de la recherche scientifique"
[3] "Laboratoire d'Automatique, Génie Informatique et Signal"
[4] "Laboratoire d'Informatique Fondamentale de Lille"
[5] "Institut de recherche sur les composants logiciels et matériels pour l'information et la communication avancée de Lille"

1.2 Research activities

CRIStAL's research activities concern topics related to the major scientific and societal issues of the moment such as: BigData, software, computer imaging, human-machine interactions, robotics, control and supervision of large systems, intelligent embedded systems, bioinformatics... The laboratry is involved in the developpment of revolutionary platforms such as Pharo, a pure object oriented language and a powerful yet simple developpment environment used worldwide.

1.3 2XS team

The 2XS (eXtra Small, eXtra Safe) team is working on highly constrained embedded devices, precisely on designing software and hardware that are secure, safe and efficient. Research in this team is focused on defining new system architectures or new languages to allow fast development of reliable embedded software. The team addresses issues concerning memory footprint, energy consumption and security and takes profit of proficiencies in formal verification, hardware/software co-design and operating system architectures to tackle the aforementioned issues.

The team is lead by *Prof.Gilles Grimaud* and has 15 members[1] as well as several trainees and most of its current work mainly revolves around the PIP project and its possible applications.

Figure 1.2: 2XS team organigram

[1] as of July 29, 2017

2. About the internship

2.1 Objectives

2.2 Coq & Coqide

Coq is the result of about 30 years of research. It started in 1984 as an implementation of the Calculus of Constructions, an expressive formal language, at INRIA by Thierry Coquand and Gérard Huet and was extended,later in 1991, by Christine to the Calculus of Inductive Constructions.

Coq is a **formal proof management system**. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the certification of properties of programming languages (e.g. the CompCert compiler certification project, or the Bedrock verified low-level programming library), the formalization of mathematics (e.g. the full formalization of the Feit-Thompson theorem or homotopy type theory) and teaching. It implements a program specification and mathematical higher-level language called Gallina that is based on the Calculus of Inductive Constructions combining both a higher-order logic and a richly-typed functional programming language.

As a **proof development system**, Coq provides interactive proof methods, decision and semi-decision algorithms as well as a tactic language letting the user define his own proof methods. Furthermore, as a **platform for the formalization of mathematics or the development of programs**, Coq provides support for high-level notations, implicit contents and various other useful kinds of macros.

For this internship, the latest version of Coq, 8.6 released in December 2016, was used. It features among other things a faster universe checker, asynchronous error processing, proof search improvements, generalized intro patterns, a new warning system, patterns in abstractions and a new subterm selection algorithm.

2.3 organization

3. PIP project

3.1 PIP protokernel

3.1.1 A minimal OS kernel with proovable isolation

An operating system is organized as a hierarchy of layers, each one constructed upon the one below it. Each layer focuses on an essential role of the operating system such as memory management, multiprogramming, input/output ... Generally speaking, while developping a kernel for an operating system based on the layered approach, the designers have a choice where to draw the kernel-user boundary. Traditionally, all the layers went in the kernel, but that is not necessary. In fact, putting as little as possible in kernel mode is safer because kernel bugs can bring down the system instantly. In contrast, user processes/layers are set up to have less power so that a bug there may not be fatal.

Various studies on bug density, relatively to the developped module size, age as well as other factors, have been conducted (e.g. Basilli and Perricone in 1984; and Ostrand and Weyuker in 2002). A ballpark figure for serious industrial systems is ten bugs per thousand lines of code. Operating systems are sufficiently buggy that computer manufacturers put reset buttons on them, something the manufacturers of cars, TV sets and stereos do not do, despite the large amount of software in these devices. Furthermore, Operating systems generally present hardware resources to applications through high-level abstractions such as (virtual) file systems.

Therfore, there are several OS kernel families such as :

- **Microkernels :** The basic idea behind a microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which, the microkernel, runs in kernel mode;
- **Exokernels :** The idea behind an exokernel is to force as few abstractions as possible on application developers, enabling them to make as many decisions as possible about hardware abstractions.

Although the closest kernel design to PIP is the exokernel, PIP does not belong to any of the kernel families featured in the state of art, but it is the

first member of a new kernel familiy, **protokernels**, as compared to most microkernels and exokernels, the TCB in PIP is even more restricted :

- Scheduling and IPC are done in user mode unlike a microkernel;
- Multiplexing is also done in user mode unlike an exokernel.

Whereas the kernel mode is only for **multi-level MMU conguration** (virtual memory) and **context switching**. This not only ensures less bugs density but also more feasibility of formal proof that will warrant the memory isolation property of the protkernel.

As a **minimal OS Kernel with provable isolation**, PIP focuses more on security and safety without sacrificing efficency and ensures memory isolation between different tasks running on the same device.PIP's algorithmic part is written in Gallina, the language of the Coq proof assistant, in a monadic style that allows direct translation into freestanding C. We will refer to this implementation in Gallina as the shallow embedding in contrast to the deep embedding introduced in section 3.4 p.15.

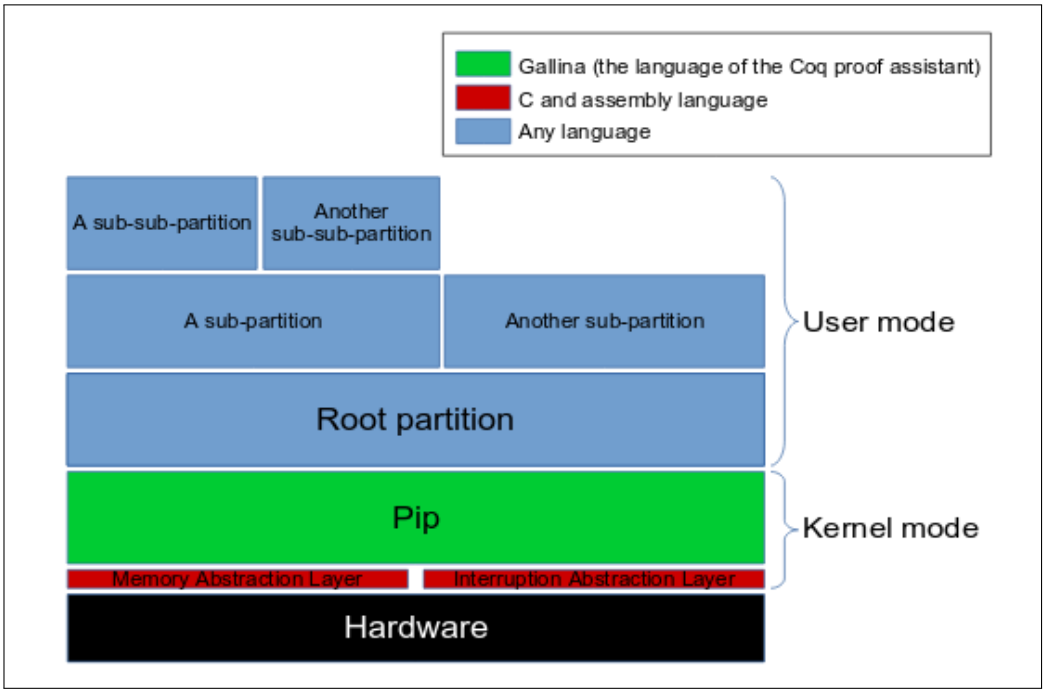

Figure 3.1: Software layers of an OS built on top of PIP

3.1.2 Horizontal isolation & vertical sharing

PIP can be used to partition the available memory which will be initially allocated to the root partition on top of PIP. Any partition can read and write in the memory of its children. However, partitions in diffrent branches of the partition tree are disjoint. The former is referred to as **vertical sharing** and the latter as **horizontal isolation**. Needless to say, all memory lended to PIP for storing kernel data, when creating partitions for exemple, are inaccessible for all partitions to prevent messing up PIP data structures which means that the kernel data is totally isolated.

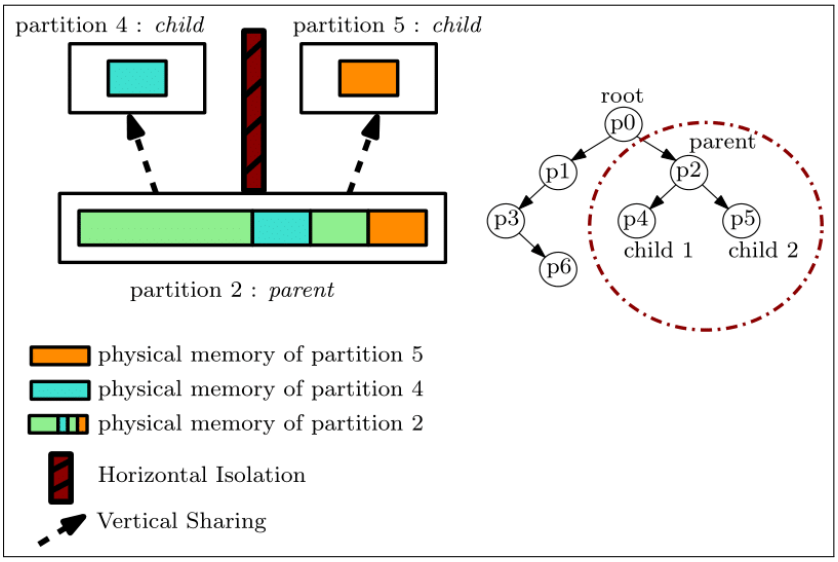

Figure 3.2: Horizontal isolation & vertical sharing in PIP

If we consider a more realistic partition tree as shown in figure 3.1.2 in which we consider Linux and FreeRTOS as sub-partitions of a root partition, multiplexer. Knowing that FreeRTOS is a real-time OS that does not isolate its tasks, we easily secured it with task isolation by porting it on PIP.

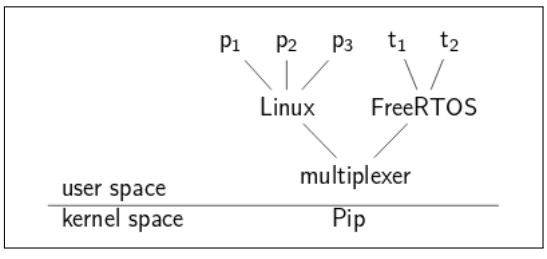

Figure 3.3: FreeRTOS task isolation using PIP

3.1.3 Proof oriented design

PIP's isolation properties are meant to be formally prooved independently of the platform it's running onto. Consequently, the algorithm and the architecture dependant part were separated. Indeed, as shown in figure 3.4, PIP is splitted into two distinct layers :

- **HAL :** gives direct access to the architecture and hardware;
- **API :** implements the algorithmic part to configure the virtual memory and the hardware.

The API code is written and proven using the Coq proof assistant, and uses the interface provided by the HAL to perform any hardware related operation. the proofs are based on Hoare logic theory introduced in section 3.2 p.13.

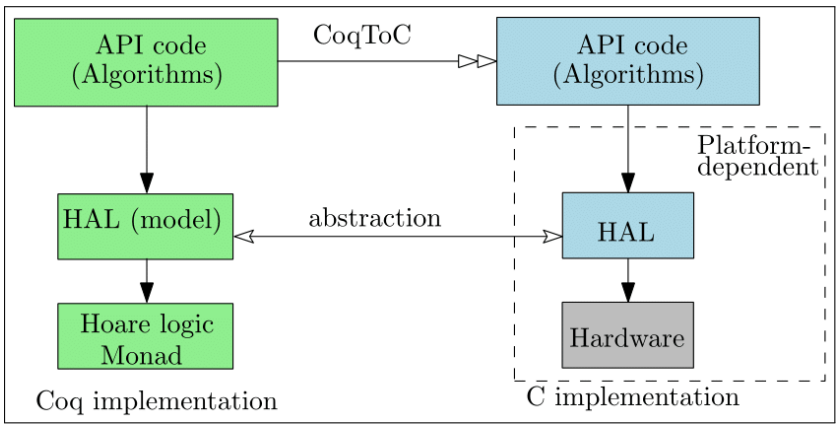

Figure 3.4: PIP design

PIP's HAL is splitted into three components, as shown in figure 3.5, each handling a specific part of the target platform's hardware :

- **Memory Abstraction Layer (MAL) :** provides an interface for the configuration of the MMU chip;
- **Interrupt abstraction Layer (IAL) :** provides an iterface to dispatch interrupts and configure hardware;
- **Bootstrap :** contains the low-level code required to boot the system.

PIP only provides system calls for management of partitions and for context switching, thus reducing the TCB to its bare minimum as explained in section 3.1.1 p.5. This user exposed API can be called by any partition using a platform dependant call :

- **createPartition :** creates a new child (sub-partition) into the current partition;
- **deletePartition :** removes a child partition and puts all its used pages back in the current partition;
- **prepare :** adds required configuration tables into a child partition to map a new virtual address;
- **collect :** removes the empty configuration tables which are not used anymore and gives it back to the current partition;
- **countToMap :** returns the amount of configuration tables needed to perform a mapping for a given virtual address;
- **addVaddr :** maps a virtual address into the given child;
- **removeVAddr :** removes a given mapping from a given child.

This API is sufficient as far as memory requirements are concerned but it lacks a way to handle interrupts. Hardware interrupts are implicitly handled by PIP and automatically dispatched to the root partition, while software interrupts, such as system calls, are notified to the partent partition of the caller and can be managed by these two additional services of PIP :

- **dispatch :** notifies an interrupt to a given partition, interrpting its current control flow and backing it up for a further resume call;
- **resume :** restores a previously interrupted context.

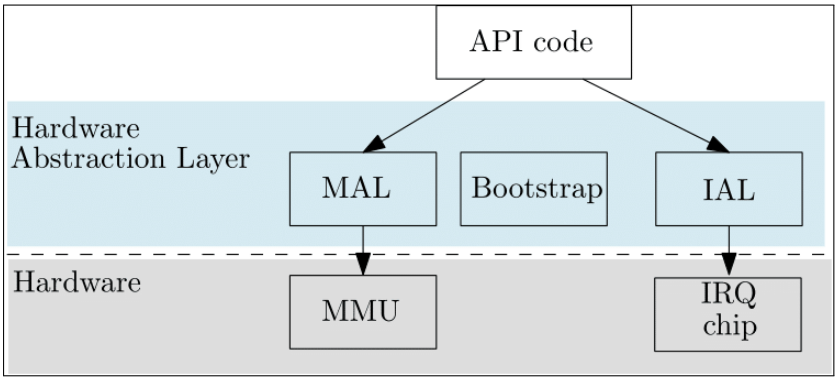

Figure 3.5: HAL and API relationship

3.1.4 More In depth understanding of PIP's Data structures

The memory

PIP uses several data stuctures per partition, which will represent the global state of the partition's memory state. This is necessary as it has to keep track of pages allocated to partitions in order to allow or deny derivation and partition creation while preserving the required properties.

Figure 3.6 shows a partition tree example consisting of a partition, Parent, that has a single child, Child1. A partition is identified by a partition descriptor which a page number essentiel to access the whole data structure of a partition. In our case, the partition descriptors of Parent and Child1 are respectively 1 and 12. The pages lent to PIP to manage a partition are organized in a tree with three branches : the MMU tables, the first shadow and the second shadow. The aim of this organization is to keep additional information about each page lent to a partition. Moreover these structures have multiple goals :

- **Access control :** the first shadow is used to avoid derivating the same page multiple times;
- **Performance :** the second shadow and the configuration list are used to quickly find the virtual address of a page without having to parse the whole virtual space when the parent partition reclaims it.

As such, adding an indirection table in the MMU configuration requires two additional pages for the shadows. Therefore, this model is estimated to require roughly thre times the amount of memory a simple virtual environment would need, but it provides nevertheless a both secure and efficient API.

To proove the isolation properties on this memory structure it is essentiel to assure it's consistency relative to the partition tree, the well typedness as well as some other consistency properties that will be detailed in section 3.2 p.13.

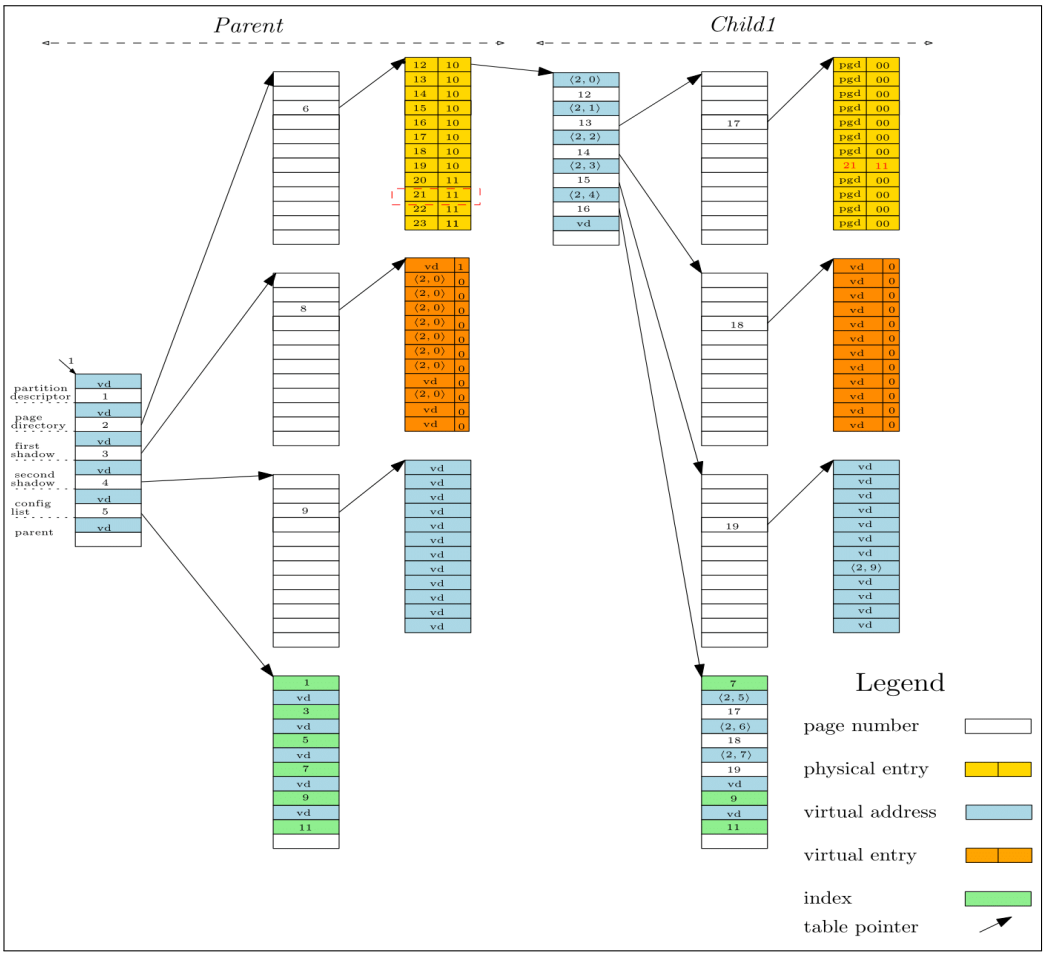

Figure 3.6: An example of a partition tree

The state

The PIP state is defined as follows :

Script 3.1: PIP state definition

```
Record state : Type :=
{ currentPartition: page ; memory: list (paddr * value) }.
```

The list in the state corresponds to the physical memory and maps physical addresses to values.A physical address is defined by a physical page number and a position into this page :

Script 3.2: paddr type definition

```
Definition paddr := page * index.
```

where pages and indexes are positive integers bounded respectively by the overall number of pages and the table size :

Script 3.3: page & index type definitions

```
Record page := { p :> nat ; Hp : p < nbPage }.
Record index := { i :> nat ;  Hi : i < tableSize }.
```

Different types of values could be stored into the physical memory by Pip so the type value is an inductive type defined as follows :

Script 3.4: value type definition

```
Inductive value : Type:=
| PE: Pentry -> value
| VE: Ventry -> value
| PP: page   -> value
| VA: vaddr  -> value
| I:  index  -> value.
```

The type Pentry which represents a phisical entry consists of a physical page number along with several flags :

Script 3.5: Pentry type definition

```
Record Pentry : Type:= {
 read:    bool ;
 write:   bool ;
 exec:    bool ;
 present: bool ;
 user:    bool ;
 pa:      page }.
```

Finally, the *Ventry* type consists of a virtual address with a unique bolean flag :

Script 3.6: Ventry type definition

```
Record Ventry : Type:= { pd: bool ; va: vaddr }.
```

with virtual addresses modeled as a list of indexes of length the number of levels of the MMU plus one :

Script 3.7: vaddr type definition

```
Record vaddr : Type :=
{ va :> list index ; Hva : length va = nbLevel + 1  }.
```

3.2 Hoare logic

3.2.1 Introduction to Hoare logic theory

How can we argue that a program is correct ? Nowadays, Building reliable software is becoming more and more difficult considering the growing scale, specifications and complexity of modern systems. Therefore, tests alone can no longer ascertain the reliability of programs especially if we're talking about critical systems. Logicians, computer scientists and software engineers have responded to these challenges by developing different kinds of techniques some of which are based on formal reasoning about properties of software and tools for helping validate these properties. One of these reasoning techniques that was used to proove PIP's properties is *Floyd–Hoare logic*, often shortened to just **Hoare Logic**. It was proposed in 1969 by the British computer scientist and logician Tony Hoare and it continues to be the subject of intensive research right up to the present day. It is not only a natural way of **writing down specifications of programs** but also a technique for **proving that programs are correct** with respect to such specifications.

Let S be a Program that we want to execute starting from a certain state s, P a predicate on the state describing the condition S relies on for correct operation and Q a predicate on the resulting state after the execution of S describing the condition S establishes after correctly running. Knowing that P is verified on s, if we proove that Q is verified after the execution of s, we can ascertain that S is partially correct. And by partial correctness of S we mean that S is correct if it terminates. Using standard Hoare logic, only partial correctness can be proven, while termination needs to be proved separately. This triple S, P and Q, written as **{P} S {Q}** , is referred to as a **Hoare triple**. The assertions P and Q are respectively referred to as the **precondiction** and the **postcondition**.

For example let's consider simple Hoare triples about an assignment command :

- $\{X = 2\}X := X + 1\{X = 3\}$: is a valid Hoare triple, that can be easily formally prooved in Coq, since the postcondiction is verified after the execution of the assign command relatively to the precondition;
- $\{X = 2\}X := X * 2\{X = 3\}$: is not a valid Hoare triple since the postcondition would not be verified after the execution of the command.

Now, let's introduce some facts and rules about Hoare triples :

1. If an assertion P implies another precondition P' of a valid Hoare triple {P'} S {Q} then {P} S {Q} is also a valid Hoare triple. This is referred to as **weaking the precondition of a Hoare triple** and can be formally defined as follows :

$$\forall \text{ S P P', (P} \rightarrow \text{P')} \rightarrow \{\text{P'}\} \text{ S } \{\text{Q}\} \rightarrow \{\text{P}\} \text{ S } \{\text{Q}\}$$

2. If we can weaken a Hoare triple, we expect it to have a **weakest precondition**. This notion was introduced by Dijkstra in 1976 and is very important since it enables us to proove total correctness and in particular program termination. Indeed, if a program doesn't terminate, its weakest precondition would be *True* and it would verify any postcondition.

3. If we consider a language containing a **SKIP instruction** which practically does nothing, we can affirm that this command preserves any property which means :

$$\forall \text{ P, } \{\text{P}\} \text{ SKIP } \{\text{P}\}$$

4. If we consider a language that allows **assingments**, in the form of X := a, then we can conlude that an arbitrary property Q holds after such assignment if we assume Q[X→a] which means Q with all occurences of X replaced by a :

$$\forall \text{ Q X a, } \{\text{Q}[\text{X}\rightarrow\text{a}]\} \text{ X:= a } \{\text{Q}\}$$

5. Generally speaking, every Program is built using **sequencing** of commands that we will write as C1;C2. Our aim is to proove a Hoare triple on this sequence with P and Q respectively as the precondition and the postcondition of this triple. This requires prooving that C1 takes any state where P holds to a state where an intermediate assertion I holds and C2 takes any state where I holds to one where Q holds which could be formally stated as :

$$\forall \text{ P I Q C1 C2, } \{\text{P}\} \text{ C1 } \{\text{I}\} \rightarrow \{\text{I}\} \text{ C2 } \{\text{Q}\} \rightarrow \{\text{P}\} \text{ C1;C2 } \{\text{Q}\}$$

3.3 Verification of invariants in the shallow embedding

3.4 The deep embedding

3.5 Hoare logic in the deep embedding