Verifying user-space systems

Matt Brecknell

Kry10 Limited

seL4 Summit – October 2024 – Sydney

Wanted

Foundational verification of deep properties of dynamic systems comprised of trusted and untrusted components

Foundational verification	Machine-checked proofs in mathematical logic
Deep properties	Functional correctness, integrity
Dynamic systems	Concurrency, interaction between components
Trusted components	Properties depend on component behaviour
Untrusted components	Robust in the presence of faulty or malicious code

To reason about systems of interacting components, we must exploit structure

- Separation of concerns
- Modularity
- Abstraction
- Compositional reasoning

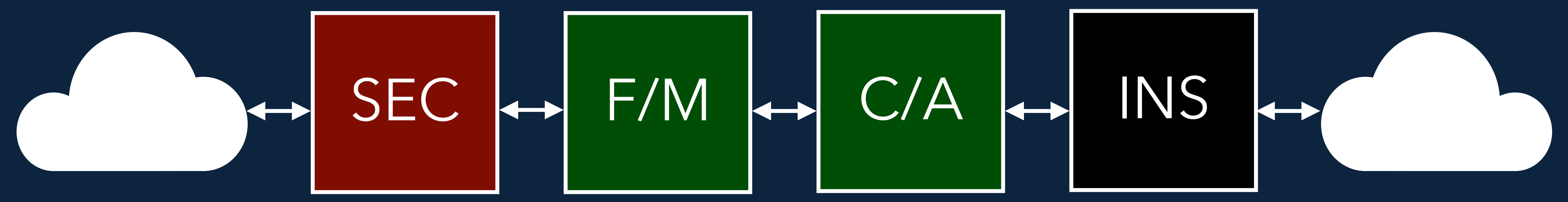
Automate within structure

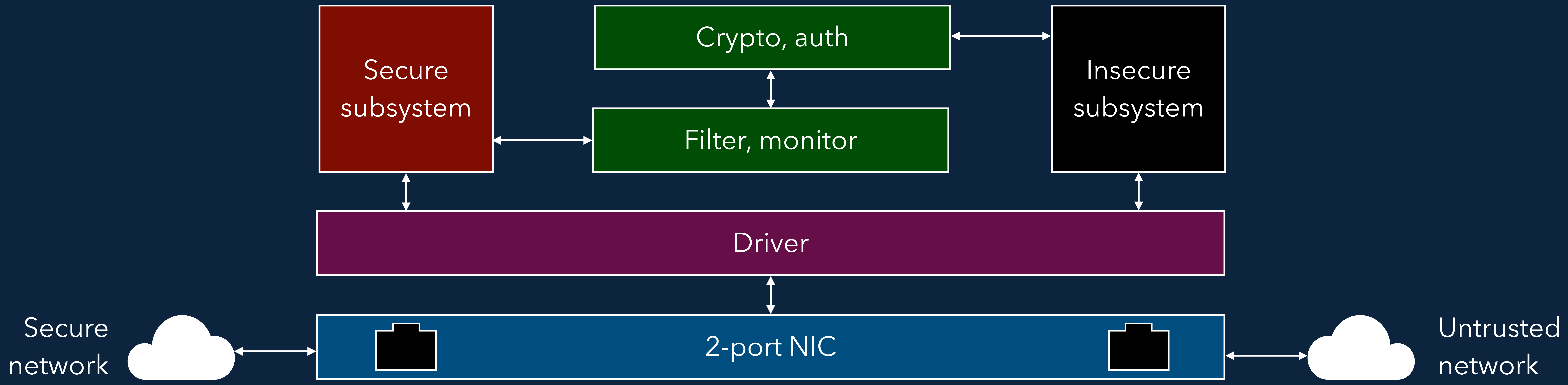
Example system – ICS gateway

Informal requirement

Traffic between the secure subsystem and the untrusted network must be encrypted, authenticated, filtered and monitored

Formal specification (process model)





Example system – sDDF

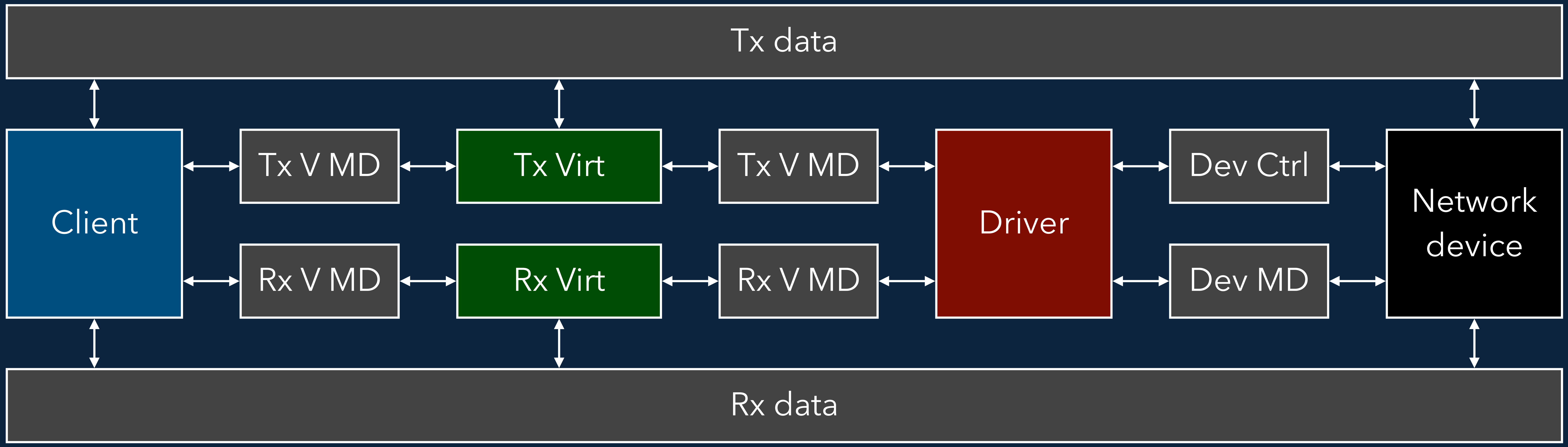
Informal requirement

Packets should only be delivered to the intended network address (Tx) or client (Rx)

Internal invariant

Every data region has a unique owner

High-level specification (process model)



Source: https://trustworthy.systems/projects/drivers/sddf-design.pdf

First steps

1. Literature review
2. Experiments

Source: https://ilyasergey.net/assets/other/CSL-Family-Tree.pdf

1969: Hoare logic

Specification “triple”

`{ P } c { Q }`

`c` – Program fragment

`P` – Precondition
`Q` – Postcondition
} Predicates on global state

IF `c` starts executing in a state satisfying `P`

THEN the final state satisfies `Q`

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1969: Hoare logic

Specification "triple"

```
{ P } c { Q }
```

`c` – Program fragment

`P` – Precondition
`Q` – Postcondition

$\left.\begin{array}{l}\texttt{P} \text{ – Precondition} \\ \texttt{Q} \text{ – Postcondition}\end{array}\right\}$ Predicates on global state

IF `c` starts executing in a state satisfying `P`

THEN the final state satisfies `Q`

Deduction rule for sequenced programs

$$\frac{\{\ P\ \}\ c_1\ \{\ Q\ \} \qquad \{\ Q\ \}\ c_2\ \{\ R\ \}}{\{\ P\ \}\ c_1\ ;\ c_2\ \{\ R\ \}}$$

Local, but might not compose

```
{ λh. h[l] = v }
  l ← !l + 1
{ λh. h[l] = v + 1 }
```

Composable, but not local

```
{ λh. Q (h[l := h[l]]) }
  l ← !l + 1
{ λh. Q h }
```

2002: Separation logic

Separating conjunction

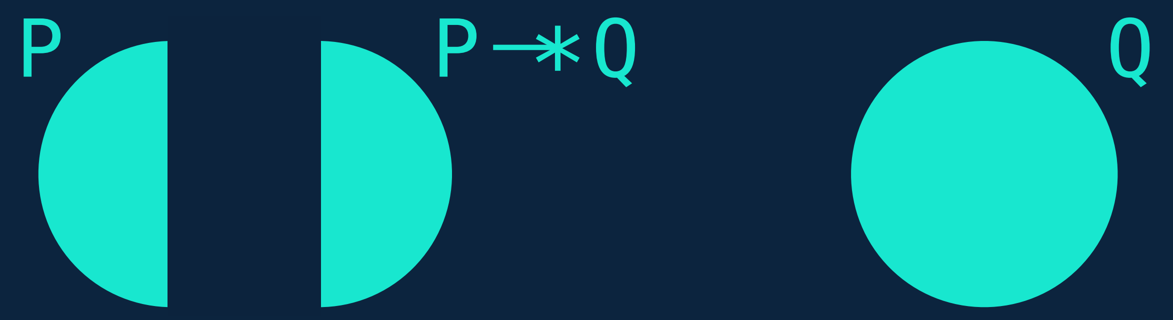
P ∗ Q

Satisfied by a resource if it can be partitioned so that

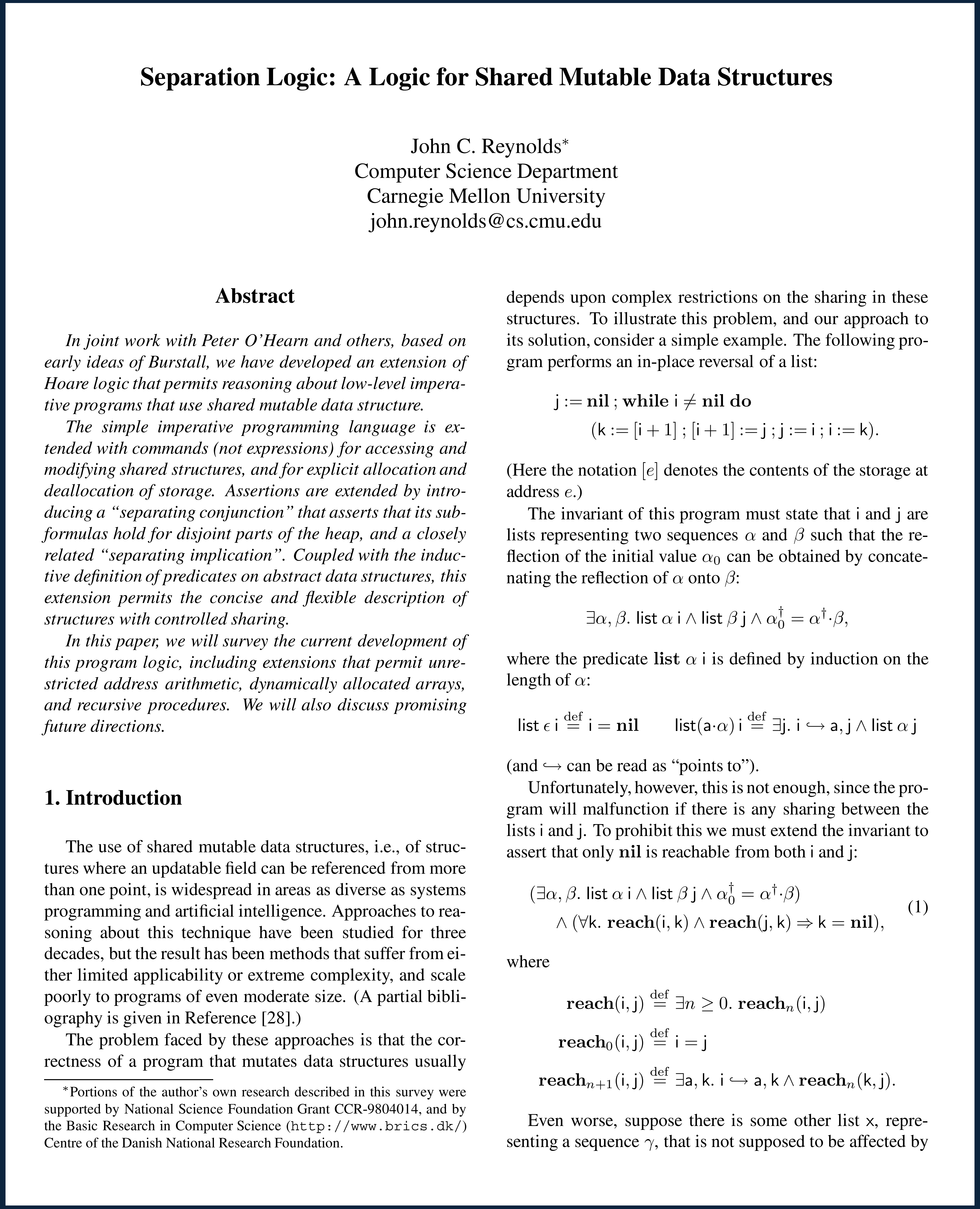
- P is satisfied by one part
- Q is satisfied by the other

Separating implication

P −∗ Q



Satisfied by a resource if adding any separate resource satisfied by P gives a resources that satisfies Q



Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds*
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

In joint work with Peter O'Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure.

The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a "separating conjunction" that asserts that its subformulas hold for disjoint parts of the heap, and a closely related "separating implication". Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing.

In this paper, we will survey the current development of this program logic, including extensions that permit unrestricted address arithmetic, dynamically allocated arrays, and recursive procedures. We will also discuss promising future directions.

1. Introduction

The use of shared mutable data structures, i.e., of structures where an updatable field can be referenced from more than one point, is widespread in areas as diverse as systems programming and artificial intelligence. Approaches to reasoning about this technique have been studied for three decades, but the result has been methods that suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size. (A partial bibliography is given in Reference [28].)

The problem faced by these approaches is that the correctness of a program that mutates data structures usually depends upon complex restrictions on the sharing in these structures. To illustrate this problem, and our approach to its solution, consider a simple example. The following program performs an in-place reversal of a list:

$$\mathsf{j} := \mathbf{nil}\ ;\ \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}\ (\mathsf{k} := [\mathsf{i}+1]\ ;\ [\mathsf{i}+1] := \mathsf{j}\ ;\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}).$$

(Here the notation $[e]$ denotes the contents of the storage at address e.)

The invariant of this program must state that i and j are lists representing two sequences α and β such that the reflection of the initial value α_0 can be obtained by concatenating the reflection of α onto β:

$$\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta,$$

where the predicate **list** α i is defined by induction on the length of α:

$$\mathsf{list}\ \epsilon\ \mathsf{i} \stackrel{\text{def}}{=} \mathsf{i} = \mathbf{nil} \qquad \mathsf{list}(\mathsf{a}{\cdot}\alpha)\ \mathsf{i} \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{j} \wedge \mathsf{list}\ \alpha\ \mathsf{j}$$

(and $\hookrightarrow$ can be read as "points to").

Unfortunately, however, this is not enough, since the program will malfunction if there is any sharing between the lists i and j. To prohibit this we must extend the invariant to assert that only **nil** is reachable from both i and j:

$$(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta) \wedge (\forall \mathsf{k}.\ \mathbf{reach}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reach}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}), \tag{1}$$

where

$$\mathbf{reach}(\mathsf{i},\mathsf{j}) \stackrel{\text{def}}{=} \exists n \geq 0.\ \mathbf{reach}_n(\mathsf{i},\mathsf{j})$$

$$\mathbf{reach}_0(\mathsf{i},\mathsf{j}) \stackrel{\text{def}}{=} \mathsf{i} = \mathsf{j}$$

$$\mathbf{reach}_{n+1}(\mathsf{i},\mathsf{j}) \stackrel{\text{def}}{=} \exists \mathsf{a}, \mathsf{k}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{k} \wedge \mathbf{reach}_n(\mathsf{k},\mathsf{j}).$$

Even worse, suppose there is some other list x, representing a sequence γ, that is not supposed to be affected by

*Portions of the author's own research described in this survey were supported by National Science Foundation Grant CCR-9804014, and by the Basic Research in Computer Science (http://www.brics.dk/) Centre of the Danish National Research Foundation.

2002: Separation logic

Separating conjunction

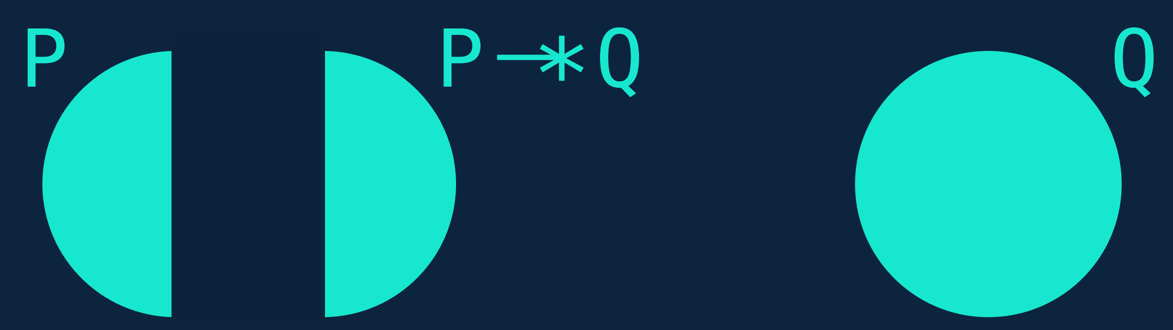
$P * Q$

Satisfied by a resource if it can be partitioned so that

- P is satisfied by one part
- Q is satisfied by the other

Separating implication

$P \mathbin{-\!\!*} Q$



Satisfied by a resource if adding any separate resource satisfied by P gives a resources that satisfies Q

Heap resources

$l \mapsto v$

Satisfied by a partial heap with

- value v at location l

$l_1 \mapsto v_1 \ * \ l_2 \mapsto v_2$

Satisfied by a heap with

- distinct locations l_1 and l_2
- value v_1 at location l_1
- value v_2 at location l_2

2002: Separation logic

$\{\ P\ \}\ c\ \{\ Q\ \}$

IF c starts executing in a state satisfying P

THEN
- c will not fail
- the final state satisfies Q

Frame rule

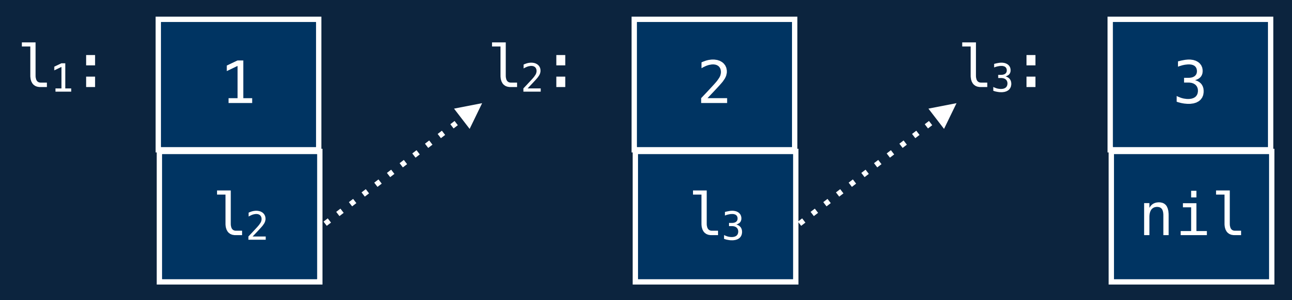
$$\frac{\{\ P\ \}\ c\ \{\ Q\ \}}{\{\ P * R\ \}\ c\ \{\ Q * R\ \}}$$

2002: Separation logic

Representation predicate – singly-linked list

```
list :: Loc → [Val] -> Prop
list hd [] = (hd = nil)
list hd (x:xs) = ∃ next. hd ↦ [x,next] ∗ list next xs
```

```
list l₁ [1,2,3]
```




```
{ list l₁ xs ∗ list l₂ ys }
  splice l₁ l₂
{ list l₁ (xs ++ ys) }
```

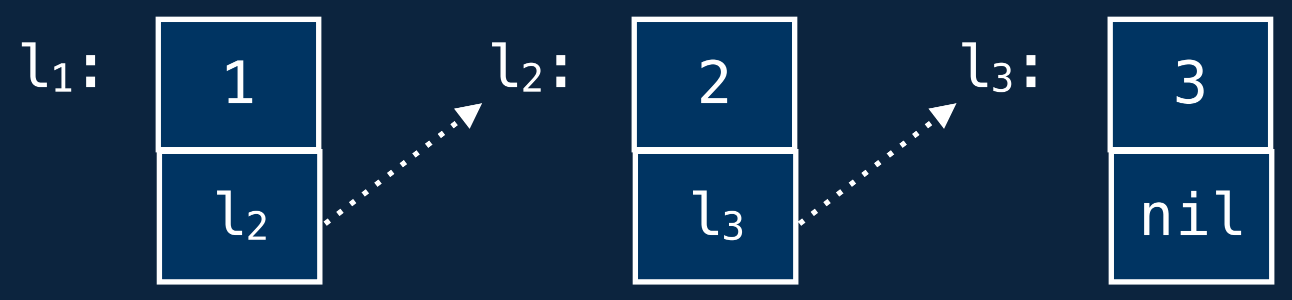
```
{ list l₁ xs ∗ list l₂ ys ∗ list l₃ zs }
  splice l₂ l₃ ; splice l₁ l₂
{ list l₁ (xs ++ ys ++ zs) }
```

2002: Separation logic

Representation predicate – singly-linked list

```
list :: Loc → [Val] -> Prop
list hd [] = (hd = nil)
list hd (x:xs) = ∃ next. hd ↦ [x,next] ∗ list next xs
```

```
list l₁ [1,2,3]
```




```
{ list l₁ xs ∗ list l₂ ys }
  splice l₁ l₂
{ list l₁ (xs ++ ys) }
```

```
{ list l₁ xs ∗ list l₂ ys ∗ list l₃ zs }
  splice l₂ l₃ ;
{ list l₁ xs ∗ list l₂ (ys ++ zs) }
  splice l₁ l₂
{ list l₁ (xs ++ ys ++ zs) }
```

2004: Concurrent separation logic

Parallel composition rule

$$\frac{\{\ P_1\ \}\ c_1\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ c_2\ \{\ Q_2\ \}}{\{\ P_1 * P_2\ \}\ c_1\ ||\ c_2\ \{\ Q_1 * Q_2\ \}}$$

Critical region rule

$$\frac{\{\ P * RI_l\ \}\ c\ \{\ Q * RI_l\ \}}{\{\ P\ \}\ \texttt{with}\ l\ \texttt{do}\ c\ \{\ Q\ \}}$$

One-place buffer

```
try_take lock buf r =
  with lock do
    r ← !buf ; buf ← None
```

$$RI_{lock} = (\exists x.\ buf \mapsto Some\ x) \lor (buf \mapsto None)$$

Resources, Concurrency and Local Reasoning

Peter W. O'Hearn

Queen Mary, University of London

Abstract. In this paper we show how a resource-oriented logic, separation logic, can be used to reason about the usage of resources in concurrent programs.

1 Introduction

Resource has always been a central concern in concurrent programming. Often, a number of processes share access to system resources such as memory, processor time, or network bandwidth, and correct resource usage is essential for the overall working of a system. In the 1960s and 1970s Dijkstra, Hoare and Brinch Hansen attacked the problem of resource control in their basic works on concurrent programming [8, 9, 11, 12, 1, 2]. In addition to the use of synchronization mechanisms to provide protection from inconsistent use, they stressed the importance of *resource separation* as a means of controlling the complexity of process interactions and reducing the possibility of time-dependent errors. This paper revisits their ideas using the formalism of separation logic [22].

Our initial motivation was actually rather simple-minded. Separation logic extends Hoare's logic to programs that manipulate data structures with embedded pointers. The main primitive of the logic is its separating conjunction, which allows local reasoning about the mutation of one portion of state, in a way that automatically guarantees that other portions of the system's state remain unaffected [16]. Thus far separation logic has been applied to sequential code but, because of the way it breaks state into chunks, it seemed as if the formalism might be well suited to shared-variable concurrency, where one would like to assign different portions of state to different processes.

Another motivation for this work comes from the perspective of general resource-oriented logics such as linear logic [10] and BI [17]. Given the development of these logics it might seem natural to try to apply them to the problem of reasoning about resources in concurrent programs. This paper is one attempt to do so – separation logic's assertion language is an instance of BI – but it is certainly not a final story. Several directions for further work will be discussed at the end of the paper.

There are a number of approaches to reasoning about imperative concurrent programs (e.g., [19, 21, 14]), but the ideas in an early paper of Hoare on concurrency, "Towards a Theory of Parallel Programming [11]" (henceforth, TTPP), fit particularly well with the viewpoint of separation logic. The approach there revolves around a concept of "spatial separation" as a way to organize thinking about concurrent processes, and to simplify reasoning. Based on compiler-

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 49–67, 2004.

1983: Rely guarantee

Specification 5-tuple

$R, G \vdash \{ P \} c \{ Q \}$

c – Program fragment

P – Precondition
Q – Postcondition
} Predicates on global state

R – Rely
G – Guarantee
} Transition relations on global state

IF
- c starts executing in a state satisfying P
- every atomic step by another thread is in R

THEN
- the final state satisfies Q
- every atomic step by c in G

Tentative Steps Toward a Development Method for Interfering Programs

C. B. JONES
Manchester University

Development methods for (sequential) programs that run in isolation have been studied elsewhere. Programs that run in parallel can interfere with each other, either via shared storage or by sending messages. Extensions to earlier development methods are proposed for the rigorous development of interfering programs. In particular, extensions to the specification method based on postconditions that are predicates of two states and the development methods of operation decomposition and data refinement are proposed.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Program Verification; D.3.2 [**Programming Languages**]: Language Classifications—*Ada*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Rely-conditions, guarantee-conditions, communicating sequential processes

1. INTRODUCTION

A brief review of the history of attempts to formalize the development of sequential (isolated) programs will set the context for the extensions we propose. The first results to appear were concerned with correctness proofs for complete programs and normally concentrated on trivial data structures such as natural numbers (cf. [7, 14, 31]). Subsequent papers showed how the proof rules could be used in a design process; in this way a proof could be used to justify the design step before development of the final code took place (cf. [5, 13, 39]). The wider application of such ideas became possible with the study of abstract data types and their refinement (cf. [12, 29]). The development method that evolved through [21], [20], and [18] mirrors this development but uses postconditions that are predicates of the initial and final states. This method is outlined in Section 2 below. The emphasis nowadays is more on a "rigorous method" that relies on the underlying mathematical ideas but in which these foundations are used mainly as a guide to less formal "correctness arguments." The approach of employing checklists of results (based on formal rules) as an integral part of the development

Author's address: Department of Computer Science, Manchester University, Manchester M13 9PL, England.

ACM Transactions on Programming Languages and Systems, Vol. 5, No. 4, October 1983, Pages 596–619.

Shopping list

	Hoare	SL	RG	CSL
Sequential	✔	✔	✔	✔
Concurrency			✔	✔
Fine-grained			✔	
Data-local		✔		✔
Thread-local			✔	✔

2007-2014: Hybrid logics

RGSep (Vafeiadis et al., 2007)

- Fine-grained thread-local and data-local reasoning

Deny-guarantee (Dodds et al., 2009)

- Dynamically-scoped concurrency
- Permissions as resources

CAP (Dinsdale-Young et al., 2010)
HOCAP (Svendsen et al., 2013)
iCAP (Svendsen et al., 2014)

- Data abstraction
- Protocols as transition systems
- Recursive predicates via step indexing (Appel et al., 2001)

CaReSL (Turon et al., 2013)

- Contextual refinement

TaDA (da Rocho Pinto et al., 2014)

- "Time and data abstraction"
- Logically atomic triples

2015-present: Iris

Iris (Jung et al., 2015, 2016)
(Krebbers et al., 2017, 2017)

- Modular, parametric, foundational
- Iris proof mode in Coq

ReLoC (Frumin et al., 2018)

- Contextual refinement

Actris (Hinrichsen et al., 2020)

- Message-passing with session types

DimSum (Sammler et al., 2023)

- Process algebra
- Heterogeneous systems

OCP (Swasey et al., 2017)
Cerise (Georges et al., 2021)

- Object-capability patterns
- Robustness w.r.t. untrusted code

Shopping list

	Hoare	SL	RG	CSL	Hybrids	Iris	Iris++
Sequential	✔	✔	✔	✔	✔	✔	✔
Concurrency			✔	✔	✔	✔	✔
Fine-grained			✔		✔	✔	✔
Data-local		✔		✔	✔	✔	✔
Thread-local			✔	✔	✔	✔	✔
Step indexing					✔	✔	✔
Data abstraction					✔	✔	✔
Refinement					✔	✔	✔
Logical atomicity					✔	✔	✔
Modular, extensible logic						✔	✔
Maintained implementation						✔	✔
Community						✔	✔
Process algebra							✔
Object capabilities							✔

The dilemma

Iris is a state-of-the-art framework, with an active community,
but it is implemented in the Coq prover,
while seL4 is specified and verified in Isabelle/HOL

1. Implement and maintain a framework in Isabelle/HOL

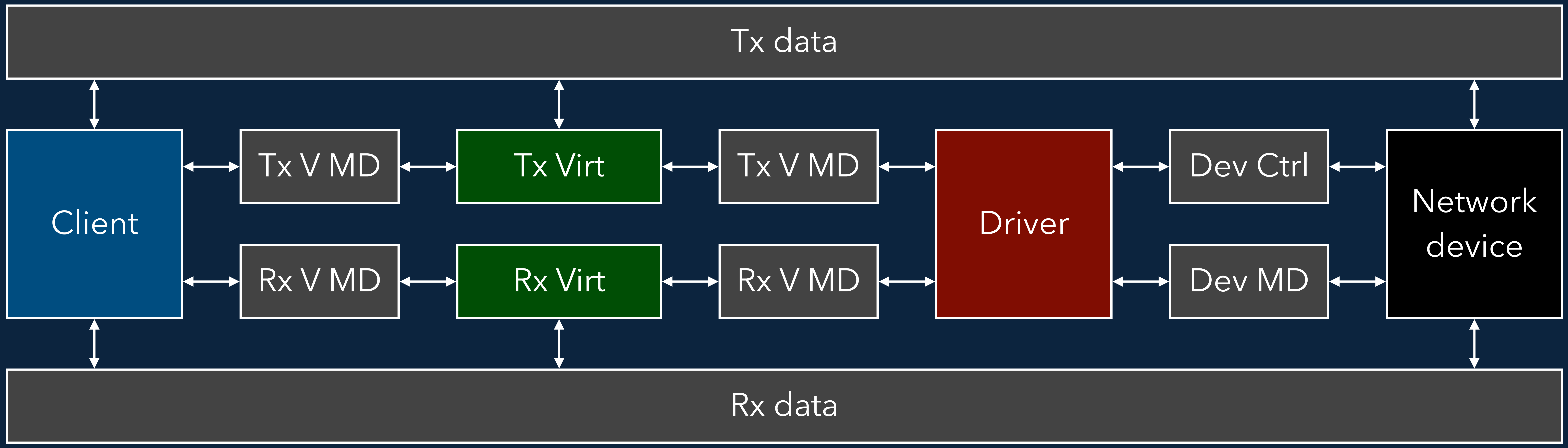
   What features do we need?
   Do their implementations port to classical logic?

2. Develop and maintain a duplicate seL4 specification in Iris

   Can we mechanise the translation?
   How will we argue that the translation is correct?

Experiments

1. Prove functional correctness of a simple (sub)system in Iris
    - sDDF network pipeline (driver, virtualiser, client)
    - Contextual refinement between
        - An abstract process model (packets are messages)
        - A concurrent intermediate specification (packets are bytes in data regions)
    - How to instantiate Iris?



Experiments

1. Prove functional correctness of a simple (sub)system in Iris
    - sDDF network pipeline (driver, virtualiser, client)
    - Contextual refinement between
        - An abstract process model (packets are messages)
        - A concurrent intermediate specification (packets are bytes in data regions)
    - How to instantiate Iris?

2. Investigate implementing a framework in Isabelle/HOL

3. Investigate transporting simple specifications between Isabelle/HOL and Iris

    Generate Isabelle/HOL, or generate Iris?

Conclusion

- Modern concurrency verification frameworks are capable and mature
- We need experiments to understand how to apply them to system-level verification
- And to seL4-based systems in particular