PROVE & RUN

77, avenue Niel, 75017 Paris, France

contact@provenrun.com

Méthodes formelles

et

Sécurité mobile

Dominique Bolignano

Prove & Run

Paris

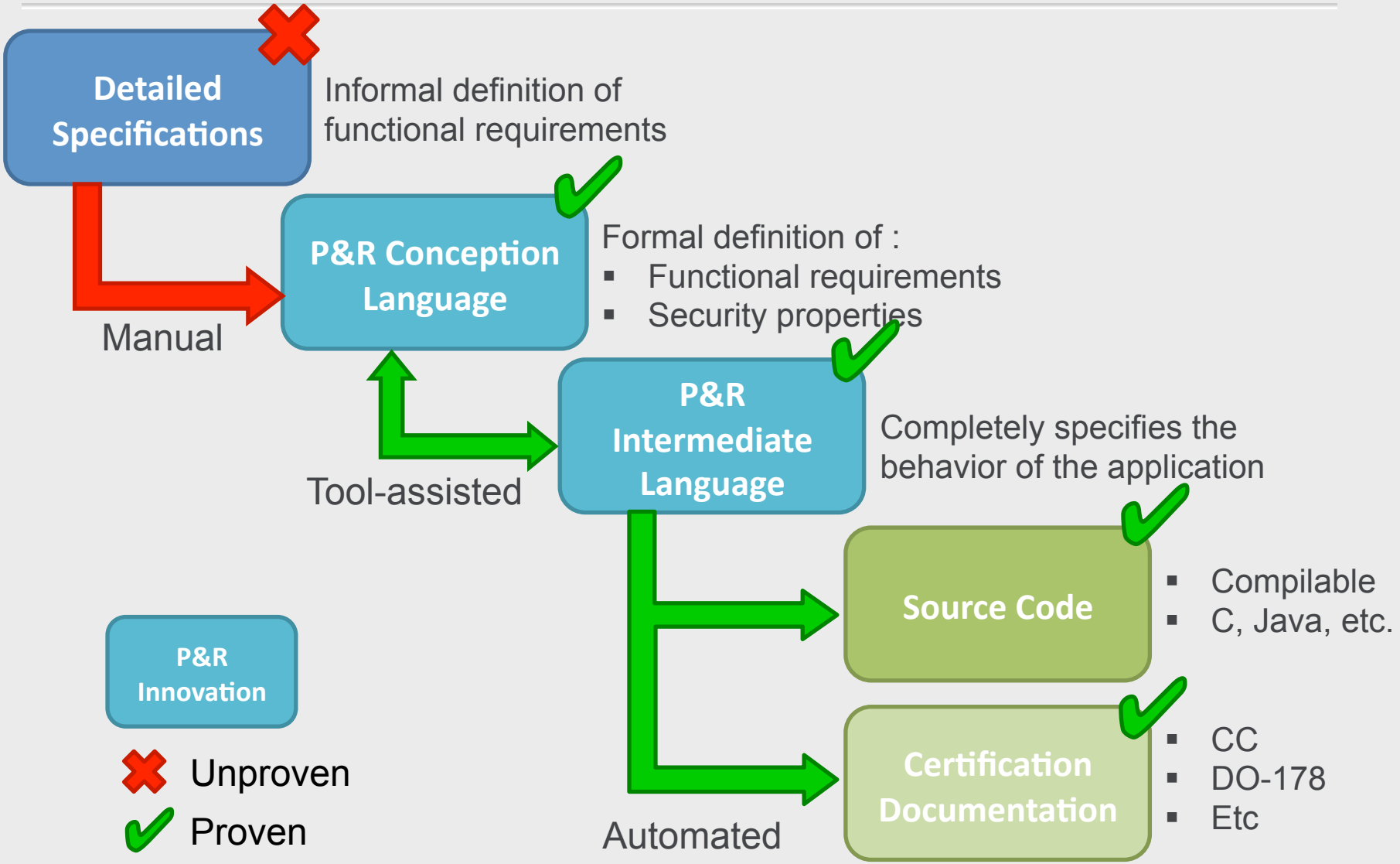
The context

- **Past experience of applying formal methods**
  - **Application in various areas of Hardware, OS, and protocol design,**
    - Architecture or protocol modelling
  - **Trusted Logic**
    - Smartcards and secure token OS and application software stack
  - **Trusted Labs**
    - Formal methods for security (service)
  - **TrustonIC**
    - Trusted Execution Environment (TEE)
  - **Prove & Run**
    - Formal methods in the large, first applied to micro-kernels, hypervisors, TEE.

Strategy

- **Choose area where cost/benefit is acceptable, but still a big and representative market.**
- **Augment benefit: where assurance becomes key,**
  - **Mobile security**
    - Importance of software attacks,
    - Bring significant confidence,
    - Formal verification of TCB,
    - Identify and correctly formalize important security properties
  - **Aerospace software**
  - **Automotive industry**
  - **Medical systems, smart grid, Automation software, …**
- **Reduce cost (micro-kernel, …).**
  - **Properly indentifying and/or defining TCB.**
  - **Benefit from reuse,**
- **Allow for maintainability,**
- **Facilitate usage of FM so it can be put into the hands of developers,**
- **Allow for certifiability,**

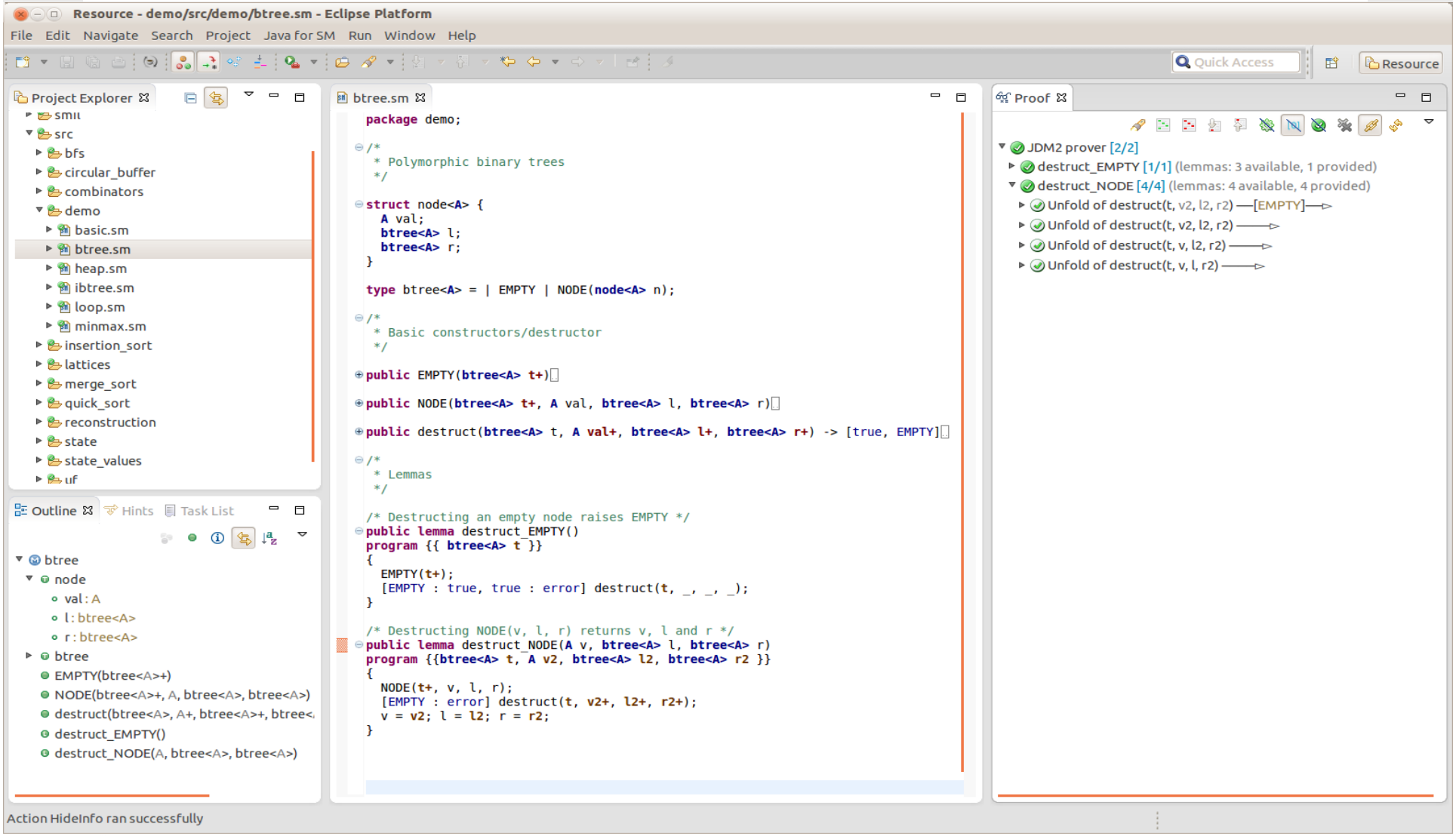
Chain


Detailed Specifications
Informal definition of functional requirements
P&R Conception Language
Formal definition of :
Functional requirements
Security properties
Manual
Tool-assisted
P&R Intermediate Language
Completely specifies the behavior of the application
Source Code
Compilable
C, Java, etc.
Certification Documentation
CC
DO-178
Etc
Automated
P&R Innovation
Unproven
Proven

Language

- Functional but with an imperative style,
- Amenable for different abstraction levels,
- Non determinism,
- Functions not necessarily total,
- Possibility of associating proof goals to paths in the control graph.
  - Proof exercise similar to symbolic debugging,
  - Facilitates certifiability,
- Properties can be expressed as tests,
  - Invariants also can be expressed as logic properties or tests,
- Possibility to use models for expressing proofs,

Language and Proof Environment



Language and Proof Environment

btree.sm

```
/*
 * Polymorphic binary trees
 */

struct node<A> {
  A val;
  btree<A> l;
  btree<A> r;
}

type btree<A> = | EMPTY | NODE(node<A> n);

/*
 * Basic constructors/destructor
 */

public EMPTY(btree<A> t+)

public NODE(btree<A> t+, A val, btree<A> l, btree<A> r)

public destruct(btree<A> t, A val+, btree<A> l+, btree<A> r+) -> [true, EMPTY]

/*
 * Lemmas
 */

/* Destructing an empty node raises EMPTY */
public lemma destruct_EMPTY()
program {{ btree<A> t }}
{
  EMPTY(t+);
  [EMPTY : true, true : error] destruct(t, _, _, _);
}

/* Destructing NODE(v, l, r) returns v, l and r */
public lemma destruct_NODE(A v, btree<A> l, btree<A> r)
program {{btree<A> t, A v2, btree<A> l2, btree<A> r2 }}
{
  NODE(t+, v, l, r);
  [EMPTY : error] destruct(t, v2+, l2+, r2+);
  v = v2; l = l2; r = r2;
}
```

Proof

- JDM2 prover [2/2]
  - destruct_EMPTY [1/1] (lemmas: 3 available, 1 provided)
  - destruct_NODE [4/4] (lemmas: 4 available, 4 provided)
    - Unfold of destruct(t, v2, l2, r2) —[EMPTY]→
    - Unfold of destruct(t, v2, l2, r2) →
    - Unfold of destruct(t, v, l2, r2) →
    - Unfold of destruct(t, v, l, r2) →

Language and Proof Environment

btree.sm

```
/*
 * Polymorphic binary trees
 */

struct node<A> {
  A val;
  btree<A> l;
  btree<A> r;
}

type btree<A> = | EMPTY | NODE(node<A> n);

/*
 * Basic constructors/destructor
 */

public EMPTY(btree<A> t+)

public NODE(btree<A> t+, A val, btree<A> l, btree<A> r)

public destruct(btree<A> t, A val+, btree<A> l+, btree<A> r+) -> [true, EMPTY]

/*
 * Lemmas
 */

/* Destructing an empty node raises EMPTY */
public lemma destruct_EMPTY()
program {{ btree<A> t }}
{
  EMPTY(t+);
  [EMPTY : true, true : error] destruct(t, _, _, _);
}

/* Destructing NODE(v, l, r) returns v, l and r */
public lemma destruct_NODE(A v, btree<A> l, btree<A> r)
program {{btree<A> t, A v2, btree<A> l2, btree<A> r2 }}
{
  NODE(t+, v, l, r);
  [EMPTY : error] destruct(t, v2+, l2+, r2+);
  v = v2; l = l2; r = r2;
}
```

Proof

- JDM2 prover [2/2]
  - destruct_EMPTY [1/1] (lemmas: 3 available, 1 provided)
  - destruct_NODE [4/4] (lemmas: 4 available, 4 provided)
    - Unfold of destruct(t, v2, l2, r2) —[EMPTY]→
    - Unfold of destruct(t, v2, l2, r2) →
    - Unfold of destruct(t, v, l2, r2) →
    - Unfold of destruct(t, v, l, r2) →

Language and Proof Environment

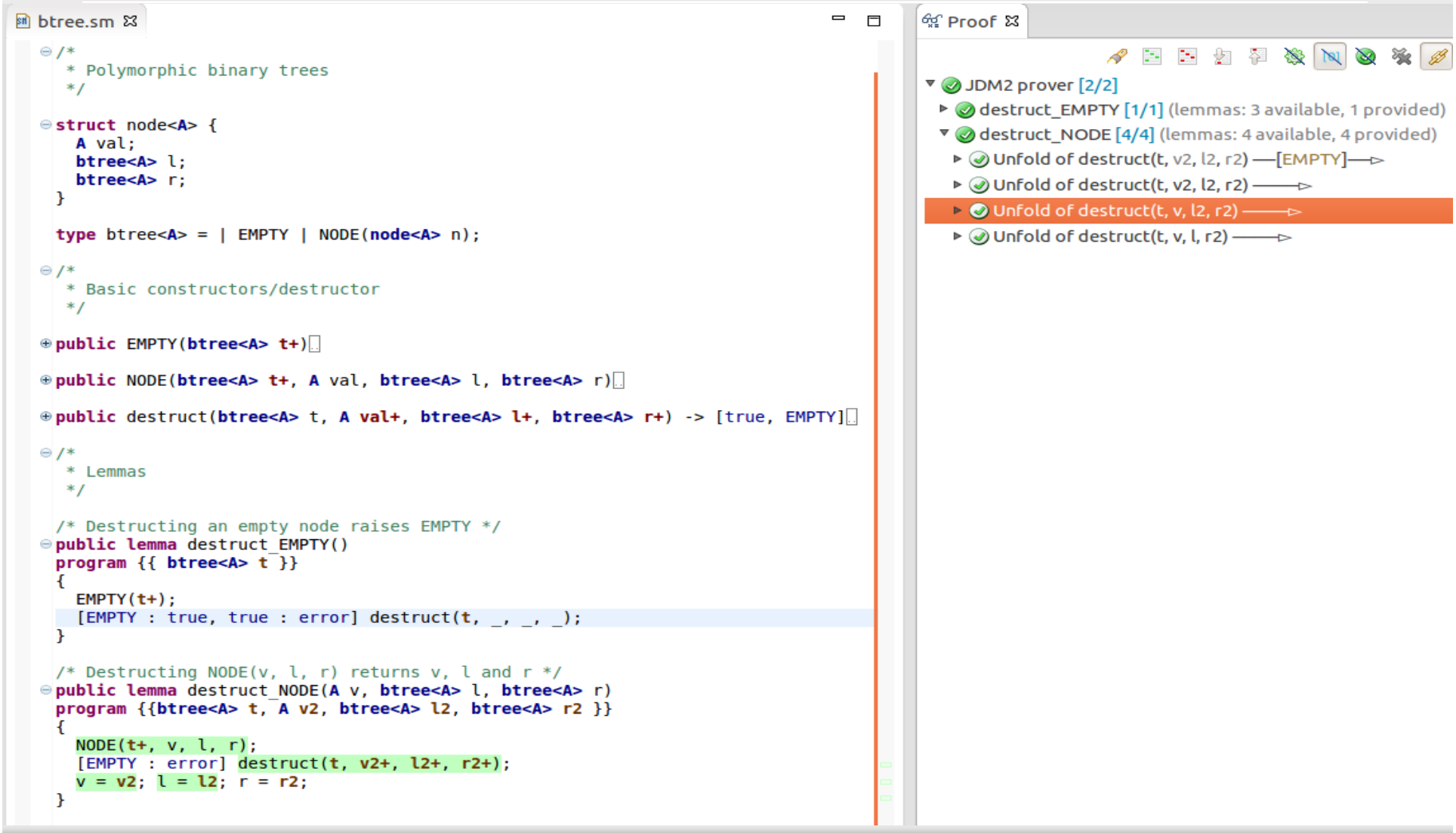
btree.sm

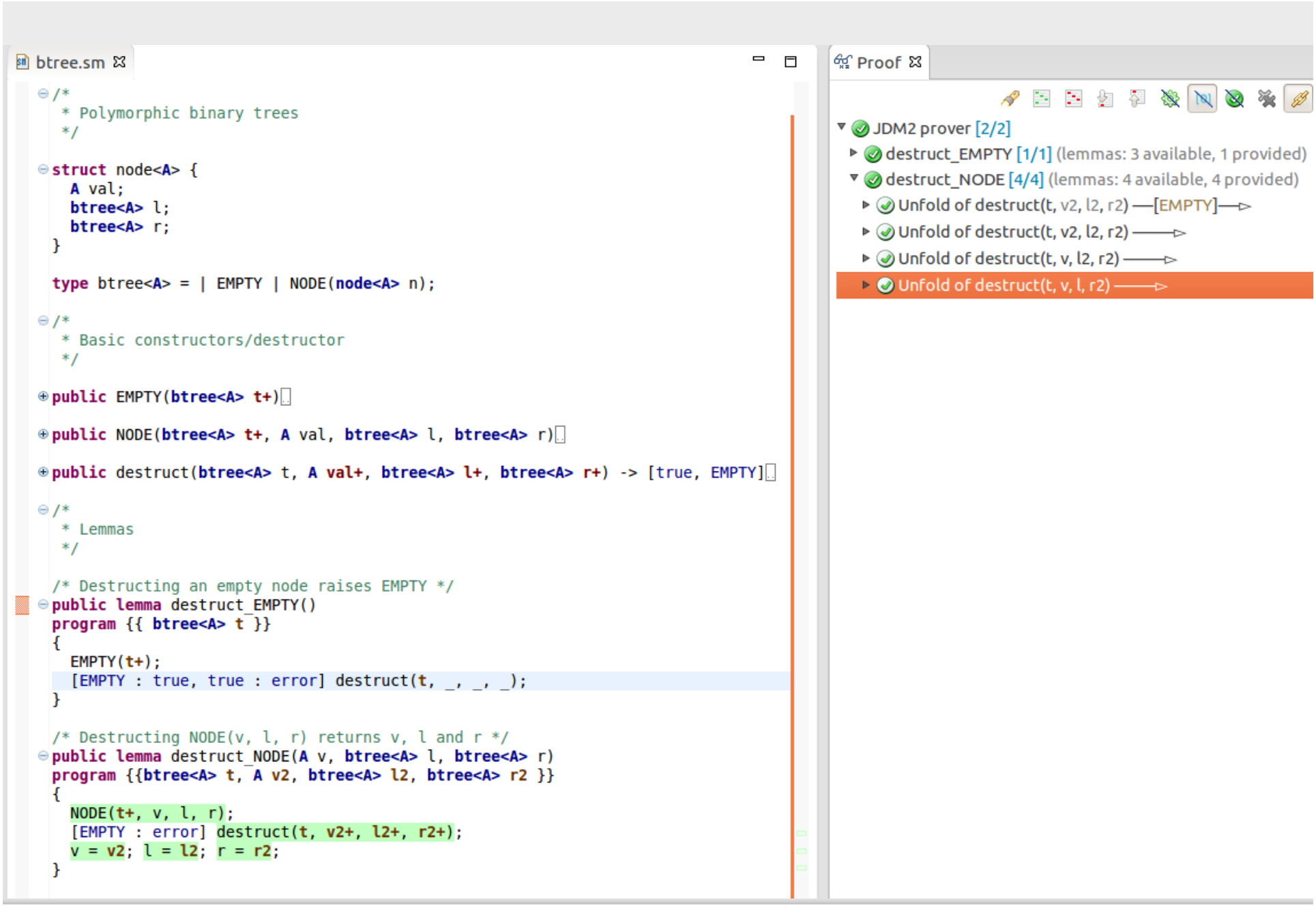
```
/*
 * Polymorphic binary trees
 */

struct node<A> {
  A val;
  btree<A> l;
  btree<A> r;
}

type btree<A> = | EMPTY | NODE(node<A> n);

/*
 * Basic constructors/destructor
 */

public EMPTY(btree<A> t+)

public NODE(btree<A> t+, A val, btree<A> l, btree<A> r)

public destruct(btree<A> t, A val+, btree<A> l+, btree<A> r+) -> [true, EMPTY]

/*
 * Lemmas
 */

/* Destructing an empty node raises EMPTY */
public lemma destruct_EMPTY()
program {{ btree<A> t }}
{
  EMPTY(t+);
  [EMPTY : true, true : error] destruct(t, _, _, _);
}

/* Destructing NODE(v, l, r) returns v, l and r */
public lemma destruct_NODE(A v, btree<A> l, btree<A> r)
program {{btree<A> t, A v2, btree<A> l2, btree<A> r2 }}
{
  NODE(t+, v, l, r);
  [EMPTY : error] destruct(t, v2+, l2+, r2+);
  v = v2; l = l2; r = r2;
}
```

Proof

- JDM2 prover [2/2]
  - destruct_EMPTY [1/1] (lemmas: 3 available, 1 provided)
  - destruct_NODE [4/4] (lemmas: 4 available, 4 provided)
    - Unfold of destruct(t, v2, l2, r2) —[EMPTY]→
    - Unfold of destruct(t, v2, l2, r2) →
    - Unfold of destruct(t, v, l2, r2) →
    - Unfold of destruct(t, v, l, r2) →

Language and Proof Environment

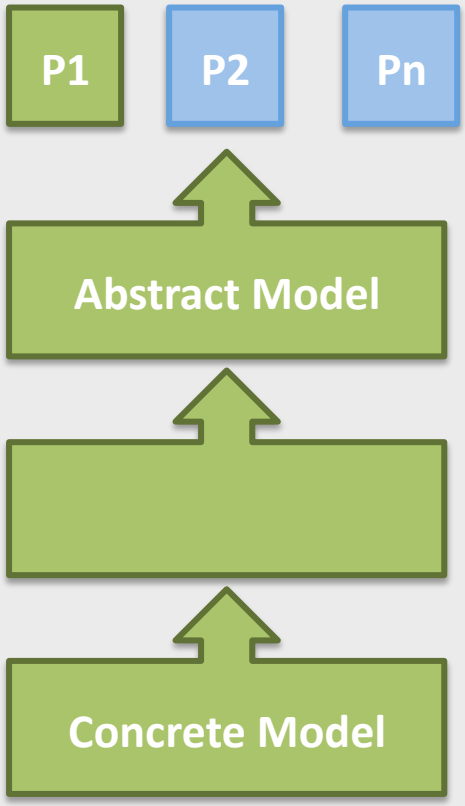



btree.sm
/*
* Polymorphic binary trees
*/
struct node<A> {
A val;
btree<A> l;
btree<A> r;
}
type btree<A> = | EMPTY | NODE(node<A> n);
/*
* Basic constructors/destructor
*/
public EMPTY(btree<A> t+)
public NODE(btree<A> t+, A val, btree<A> l, btree<A> r)
public destruct(btree<A> t, A val+, btree<A> l+, btree<A> r+) -> [true, EMPTY]
/*
* Lemmas
*/
/* Destructing an empty node raises EMPTY */
public lemma destruct_EMPTY()
program {{ btree<A> t }}
{
EMPTY(t+);
[EMPTY : true, true : error] destruct(t, _, _, _);
}
/* Destructing NODE(v, l, r) returns v, l and r */
public lemma destruct_NODE(A v, btree<A> l, btree<A> r)
program {{btree<A> t, A v2, btree<A> l2, btree<A> r2 }}
{
NODE(t+, v, l, r);
[EMPTY : error] destruct(t, v2+, l2+, r2+);
v = v2; l = l2; r = r2;
}
Proof
JDM2 prover [2/2]
destruct_EMPTY [1/1] (lemmas: 3 available, 1 provided)
destruct_NODE [4/4] (lemmas: 4 available, 4 provided)
Unfold of destruct(t, v2, l2, r2) —[EMPTY]—▷
Unfold of destruct(t, v2, l2, r2) ——▷
Unfold of destruct(t, v, l2, r2) ——▷
Unfold of destruct(t, v, l, r2) ——▷

A complete rational

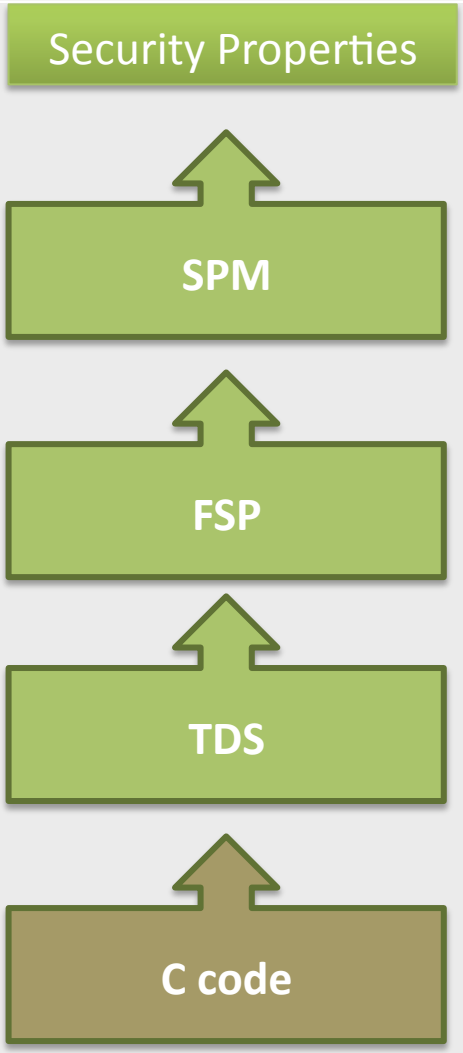
- Identify a complete rational of why we believe the system should work properly,
  - Some part of the rational being formal,
  - The rest being informal, but checkable.
- Be able to associate any problem, error, bug, to a particular error in the rational,
- The informal part should be the non risky part.

Property Expression - Assurance

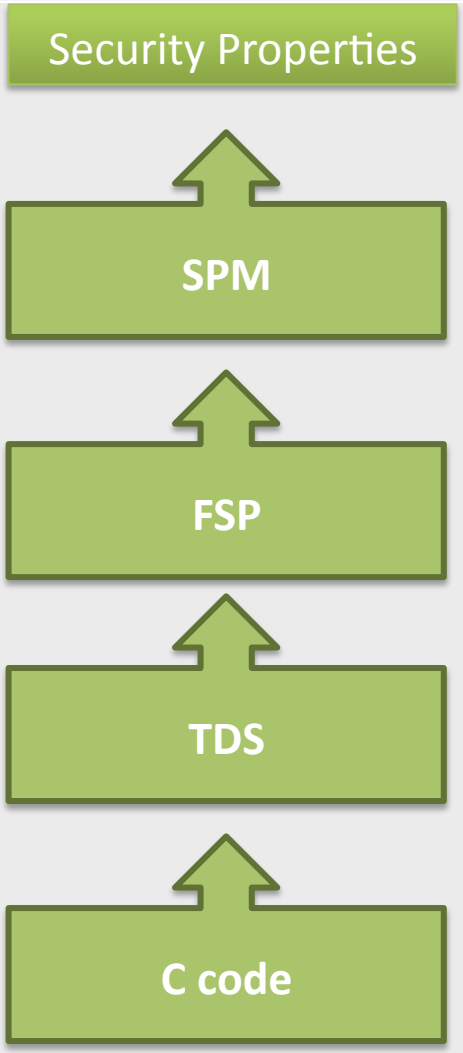


- Defining and using adequate formal properties
- Making sure these properties address targeted security needs
- In a way that is amenable to certification, at very high levels of assurance

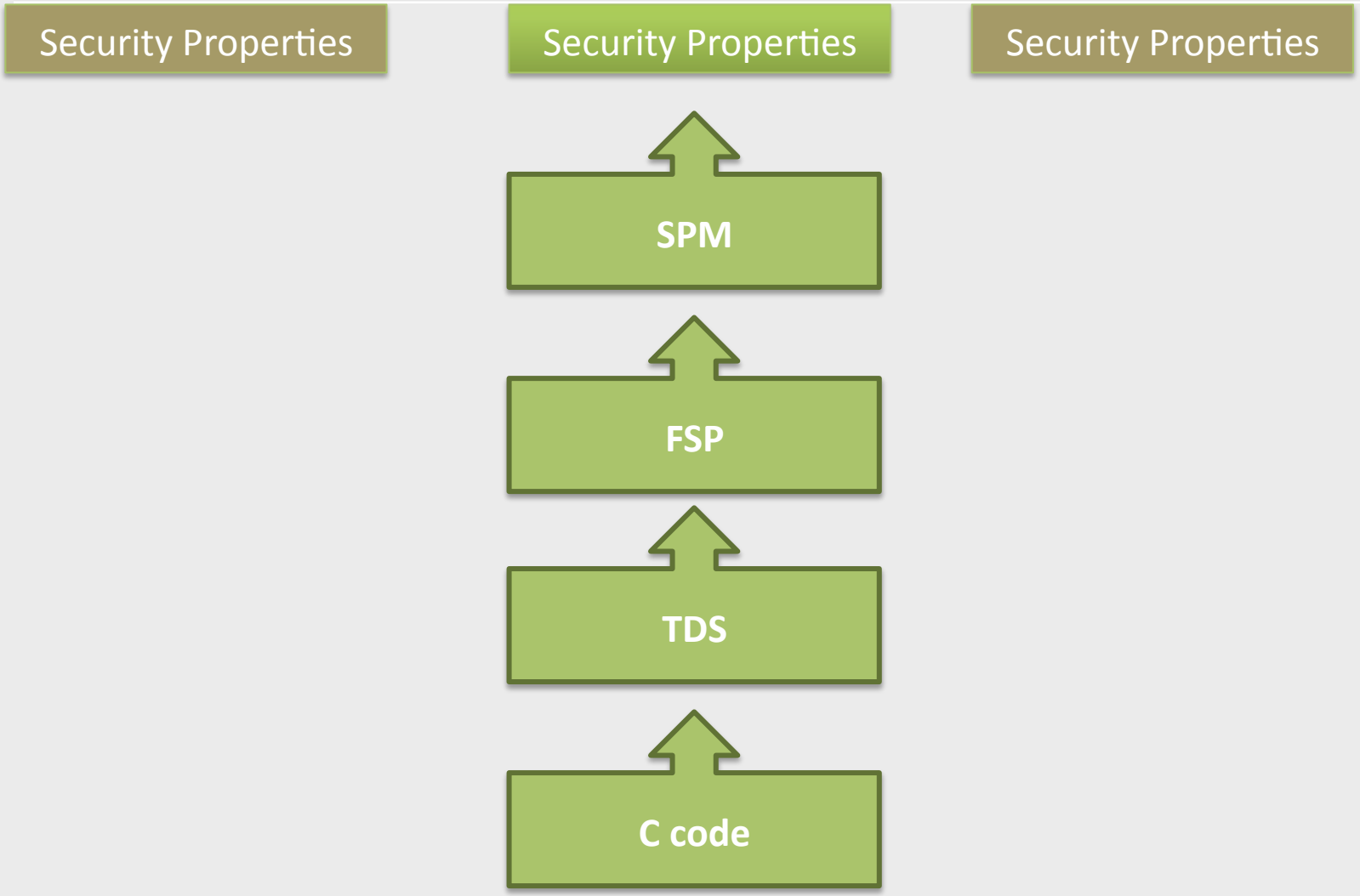
Levels of Abstractions vs CC



Levels of Abstractions vs CC



Levels of Abstractions vs CC



Why do we gain in terms of cost

- A micro-kernel type of software can be done at commercially acceptable prices,
- Proof is like debugging,
- Property expression is like test,
- There is factorization and anticipation (micro-kernel),

Why do we gain in terms of timing

- Compatible with fast prototyping when intermediate models can be executed,
- Many properties can be left unproved for some time and non completely proven program can be used and are usually still valid programs,

Why do we gain in terms of maintanability

- Some modifications have no real impact on proof principles, and are as costless as one could think of,
- Some others have impacts. These are easily located and an incremental work is done by the developer. (Some progress still to be made in some areas).

- Same is mostly true for evaluation/certification.

THANK YOU FOR YOUR TIME

QUESTIONS?

Prove & Run S.A.S.

dominique.bolignano@provenrun.com

77, avenue Niel, 75017 Paris, FRANCE