Verification of Functional Program Components[1]

Zoltán Horváth Tamás Kozsik Máté Tejfel

{hz,kto,matej}@inf.elte.hu
http://people.inf.elte.hu/{hz,kto,matej}/

Dept. of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary

NJSZT Szoftvertechnológiai Fórum, 7th February, 2007

[1] Supported by ELTE IKKK (GVOP-3.2.2-2004-07-0005/3.0) and Stiftung Aktion Österreich–Ungarn (66öu2).

Outline

Why functional programming?

- Clear program text – close to mathematical specification
- No assignments
- No side effects
- Relatively easy to prove correctness
- Ideal for trusted code

Motivation for using formal methods

- Sound concepts needed for distributed and parallel programs
- Verification of safety critical applications
- Safe usage of software components
- **Our focus**: machine verifiable mobile code

Need for trusted mobile code

Our programs often use code (applets, plug-ins etc.) written by somebody else.

- Dangers:
  - Viruses, attacks
  - Security holes in operating systems
  - Programming failures in safety critical software (embedded systems, control software of medical instruments)
  - Incomplete specifications, side effects
- We need components with proven properties
  - Resource consumption
  - Security
  - **Functionality**

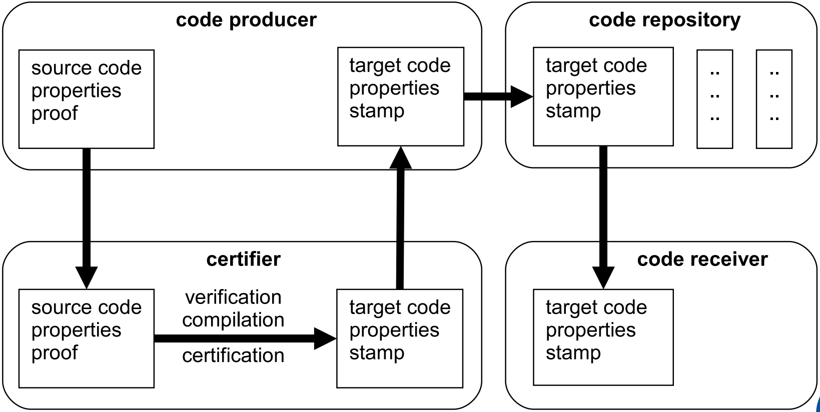
The Certified Proved-Property-Carrying Code architecture (CPPCC)

Safe mobile code exchange with minimal run-time overhead.

Three main parties involved in the scenario:

1. Producer of the mobile code: adds proofs of properties
2. Receiver: executes code only after safety checks which ensure that the code satisfies the requirements specified in the receiver's code
3. Certifying authority: reduces the work-load of the receiver, performs verification static-time

Overview of CPPCC



Our results in the FunVer project

- Extending Sparkle (the dedicated theorem prover for Clean) with support for temporal properties
- Expressing and proving temporal properties of a set of processes written in Clean
- Extending Clean dynamics with proven properties (CPPCC prototype)
- D-Clean (Distributed Clean)

Using the results

- Potential for FP in software industry
  - Embedded systems (Hume)
  - Telecommunication (Erlang)
  - FP components integrated into complex systems
- Moving results to mainstream languages / methodologies
  - C++, Java, B-method

Concepts

- Temporal properties about the states of distributed programs, for example: (subtype) invariants
- Formal proofs, machine verifiable by theorem provers
- Mobile components
  - Mobile expressions (functional code), in the FP language Clean + dynamics (Mobile Haskell, JoCaml, etc.)
  - Java Virtual Machine code
- Property/proof carrying code architecture, type and semantical checks

Foundations

- A formal model of programming is required
- The properties of the model impose constraints
  - What applications can be developed
  - What is possible to prove
  - Our model: interleaving, branching-time temporal logic

Properties of the formal model

Specification of *problems* and developing the *solutions* of problems in case of *parallel and distributed systems*.

- An extension of a relational model of non-deterministic sequential programs
- Provide tools for stepwise refinement of problems in a FP approach
- Use the concept of iterative abstract program of UNITY
- The concept of solution is based on the comparison of the problem as a relation and the (static) behaviour relation of the program

UNITY-like temporal logic

- Convenient operators
  - Safety (invariant, unless)
  - Progress (ensures, leads-to)
  - Initial and final states (init, fixed points)
- Support for component-oriented approach (Composing specifications and programs)
- Example: resource scheduling

Dining philosophers

```
:: Philo = Thinking | Hungry | Eating
```

For all *i* and *j*,

$$\neg(\mathrm{neighbours}(i,j) \wedge \mathrm{philo}_i = \mathrm{Eating} \wedge \mathrm{philo}_j = \mathrm{Eating}) \;\in\; \textbf{inv}$$

$$\mathrm{philo}_i = \mathrm{Thinking} \;\; \textbf{unless} \;\; \mathrm{philo}_i = \mathrm{Hungry}$$

$$\mathrm{philo}_i = \mathrm{Eating} \;\; \textbf{ensures} \;\; \mathrm{philo}_i = \mathrm{Thinking}$$

Composing specifications and programs

- Certain properties of a system can be computed from properties of its components
- If a statement is invariant in all components, then it is invariant in the whole application
- Ability to reason about a system
  - even if certain components are not known
  - only their properties are known
- Components received as mobile code

A concept of state in pure functional languages

- No destructive assignments, variables are constants
- Advantage: referential transparency, equational reasoning, the occurrences of the same expression have the same value
- I/O: single reference to environment, referential transparency cannot be violated, environment represented as series of pure values
- State: abstract objects corresponding to series of values

Proving invariants

To prove an invariant

- one needs to check the initial value of objects and calculate the weakest precondition for all atomic actions
- for all atomic actions we should calculate the substitution of the invariant using the state-transition function of the action
- we should prove that all these *wp*-s hold, if the invariant holds (the truth of the invariant is reserved by each action)

An unless property can be proved in a similar way, using weakest precondition calculation (rewriting).
A property "P unless(S) Q" holds if for all t atomic steps of S:
$P \wedge Q \Rightarrow wp(t, P \vee Q)$

Proving properties of communicating programs

- Example: dining philosophers
  - one server process (resource scheduler)
  - several clients (resource consumers)
- State transition: a `next_event` function (state transitions are controlled by the server, a monitor-like solution)
- From the point of view of verification we simulate the program with a `process_events` function.

State space

```
:: Philo = Thinking | Hungry | Eating
```

- Local state of a client: a value of type `Philo`
- Local state of the server: a list of `Philo`s,
- State transition: if a philosopher changes its local state, the server calculates the new local state values with the `next_event` function

State transition

```
next_event:: [Philo] Int -> ([Int],[Philo])
```

Arguments:

- the local state of the server
- the id of the client that changes its state

The result:

- the ids of the clients that can start eating
- the new local state of the server

The process_events function

Recursively calls the next_event function

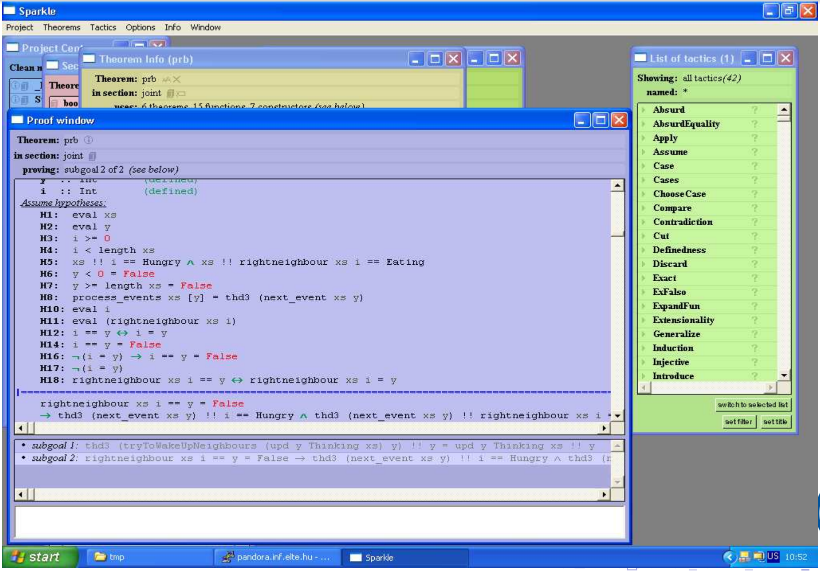
```
process_events:: [Philo] [Int] -> [Philo]

process_events philos [] = philos

process_events philos [id]
   | (id < 0) || (id >= length philos) = philos
   = snd (next_event philos id)

process_events philos [id : ids]
   # philos = process_events philos [id]
   = process_events philos ids
```

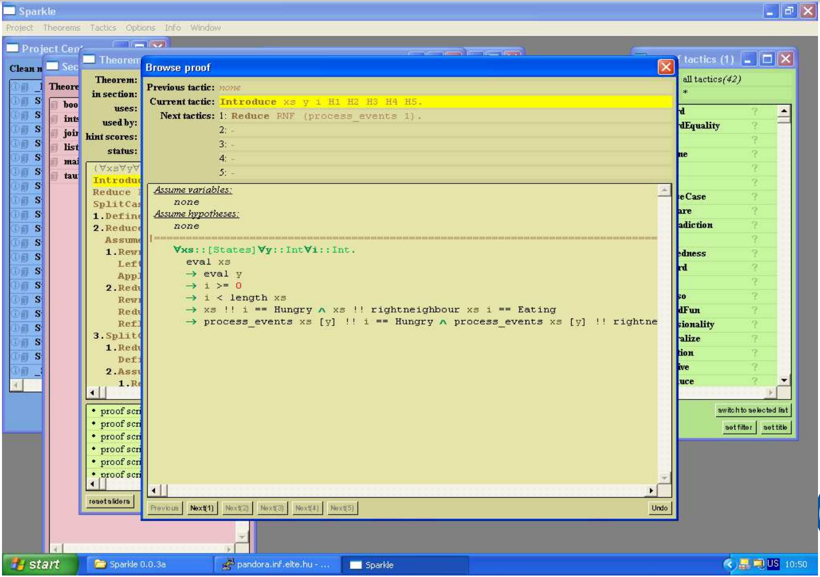
Object abstraction

- We can consider the values of the different `philos` variables as different states of the same abstract object (global state).
- For this abstract object we can formalize and prove temporal properties
- Example property: a safety property (unless) in the `process_events` function: if a client is hungry and its right neighbour is eating, then these two philos do not change state unless the neighbour starts thinking

Formalisation of an “**unless**” property

```
eval philos -> eval ids ->
(i >= 0) -> (i < length philos) ->
[
(philos!!i == Hungry) /\
   ( philos!!(rightneighbour philos i) == Eating)
UNLESS(process_events philos ids)
(philos!!(rightneighbour philos i) == Thinking)
]
```


Sparkle
Project Theorems Tactics Options Info Window
Theorem Info (prb)
Theorem: prb
in section: joint
Proof window
Theorem: prb
in section: joint
proving: subgoal 2 of 2 (see below)
i :: Int (defined)
Assume hypotheses:
H1: eval xs
H2: eval y
H3: i >= 0
H4: i < length xs
H5: xs !! i == Hungry ∧ xs !! rightneighbour xs i == Eating
H6: y < 0 = False
H7: y >= length xs = False
H8: process_events xs [y] = thd3 (next_event xs y)
H10: eval i
H11: eval (rightneighbour xs i)
H12: i == y ↔ i = y
H14: i == y = False
H16: ¬(i = y) → i == y = False
H17: ¬(i = y)
H18: rightneighbour xs i == y ↔ rightneighbour xs i = y
rightneighbour xs i == y = False
→ thd3 (next_event xs y) !! i == Hungry ∧ thd3 (next_event xs y) !! rightneighbour xs i
subgoal 1: thd3 (tryToWakeUpNeighbours (upd y Thinking xs) y) !! y = upd y Thinking xs !! y
subgoal 2: rightneighbour xs i == y = False → thd3 (next_event xs y) !! i == Hungry ∧ thd3
List of tactics (1)
Showing: all tactics (42)
named: *
Absurd
AbsurdEquality
Apply
Assume
Case
Cases
ChooseCase
Compare
Contradiction
Cut
Definedness
Discard
Exact
ExFalso
ExpandFun
Extensionality
Generalize
Induction
Injective
Introduce
switch to selected list
set filter
set title
start
tmp
pandora.inf.elte.hu - ...
Sparkle
US 10:52


Sparkle
Project Theorems Tactics Options Info Window
Browse proof
Previous tactic: none
Current tactic: Introduce xs y i H1 H2 H3 H4 H5.
Next tactics: 1: Reduce RNF (process_events 1).
Assume variables:
none
Assume hypotheses:
none
∀xs::[States]∀y::Int∀i::Int.
eval xs
→ eval y
→ i >= 0
→ i < length xs
→ xs !! i == Hungry ∧ xs !! rightneighbour xs i == Eating
→ process_events xs [y] !! i == Hungry ∧ process_events xs [y] !! rightne
Previous Next(1) Next(2) Next(3) Next(4) Next(5) Undo
start Sparkle 0.0.3a pandora.inf.elte.hu - ... Sparkle 10:50

Type system with subtype marks

Formal reasoning about properties

Combining lightweight and heavyweight tools

- Lightweight: type system
- Heavyweight: proof system

Programming language (SENYV)

- Type system supporting subtype marks
- Proof system adapted to subtype marks

Subtype marks

- Annotations attached to types
- Denote type invariants
- E.g. let S denote “sorted”
- Expressing pre- and postconditions etc.

```
Insert :: Int -> List{S} -> List{S!}
Insert e Nil = Cons e Nil
Insert e (Cons x xs) =
                  if (e <= x)
                     (Cons e (Cons x xs))
                     (Cons x (Insert e xs))
```

Semantics of subtype marks

- Typing rules for subtype mark propagation
  - used by the type system
  - very simple typing rules: easy to use for an average programmer
- Bool-functions – used by the proof system

Semantics of subtype marks (cont'd)

- Each subtype mark corresponds to a predicate
- Sparkle: Bool functions written in Clean

```
S :: !List -> Bool
S Nil = True
S (Cons x Nil) = True
S (Cons x xs=:(Cons y ys)) =
                              (x <= y) && (S xs)
```

$$S : List \to \mathbb{L}$$

$$S(list) = (S\ list = True)$$

Division of labour

Believe-me mark

```
Insert :: Int -> List{S} -> List{S!}
Insert e Nil = Cons e Nil
Insert e (Cons x xs) =
                  IfL (LessEq e x)
                      (Cons e (Cons x xs))
                      (Cons x (Insert e xs))

Sort :: List -> List{S}
Sort Nil = Nil
Sort (Cons x xs) = Insert x (Sort xs)
```

Sparkle theorem

Partial correctness of `Insert`

```
Insert :: Int -> List{S} -> List{S!}
```

$$\forall e :: \text{Int}.\ \forall xs :: \text{List}.$$
$$(xs = \bot \lor S(xs)) \rightarrow (\text{Insert } e\ xs = \bot \lor S(\text{Insert } e\ xs))$$

```
[e::Int][xs::List]
(xs = _|_ \/ S xs)
-> (Insert e xs = _|_ \/ S (Insert e xs))
```

Current work

- Subtype marks in C++ STL
- Implement subtype marks with C++ TMP

Correctness of mobile components

- Dynamically download, link and execute code
- Ensure the correctness of mobile code
- Formal reasoning is preferred
- Minimal client-side / run-time overhead

Requirements on mobile code

- It does not use too much resources
- It does not read or modify data unauthorised
- It implements the desired functionality

Solutions

- Full dynamic-time code verification just before the application of the code (static, structural and type correctness verification: well-formedness, data-flow analysis for illegal memory access, type of instruction arguments etc.)
- Trusting in the code producer unconditionally (with using a certificate mechanism, to check identity)
- Trusting in code integrity and performing run-time pattern-match for types (Clean dynamic)

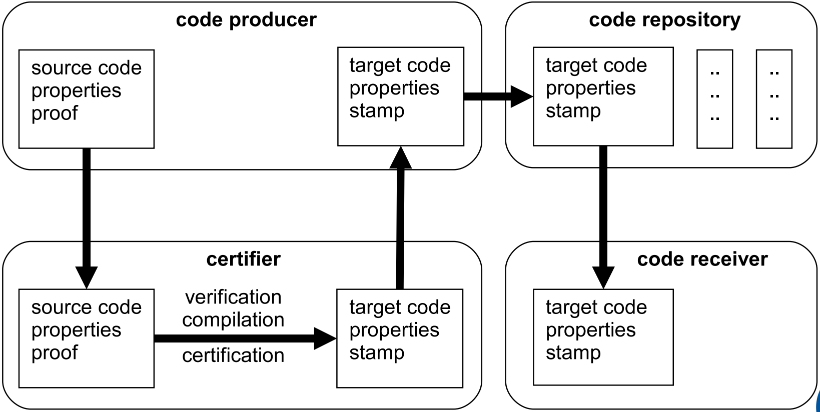
The Certified Proved-Property-Carrying Code architecture (CPPCC)

Safe mobile code exchange with minimal run-time overhead.

Three main parties involved in the scenario:

1. Producer of the mobile code: adds proofs of properties
2. Receiver: executes code only after safety checks which ensure that the code satisfies the requirements specified in the receiver's code
3. Certifying authority: reduces the work-load of the receiver, performs verification static-time

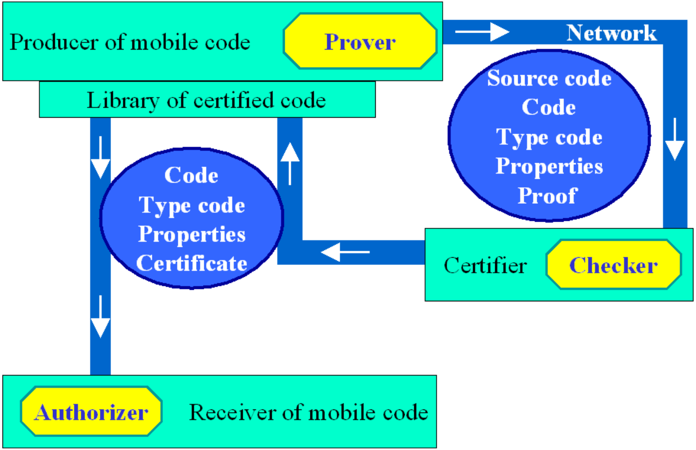
CPPCC overview



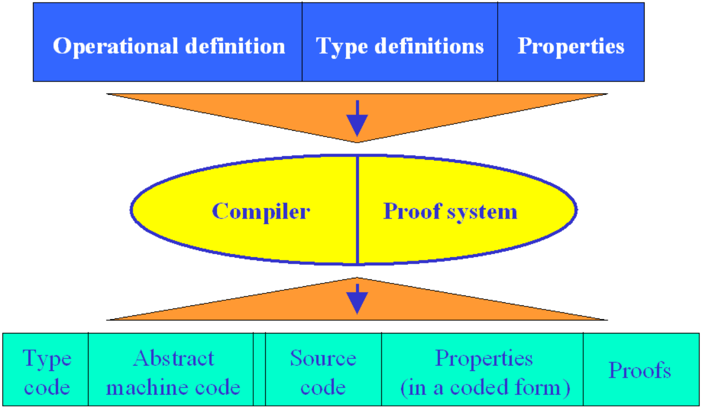
Example

- Receiver: an application using resources
- Mobile code: resource scheduler (dining philosophers)

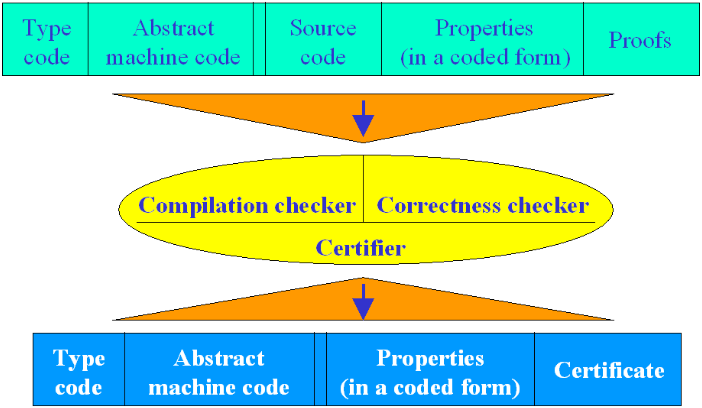
Transmission of verified mobile code



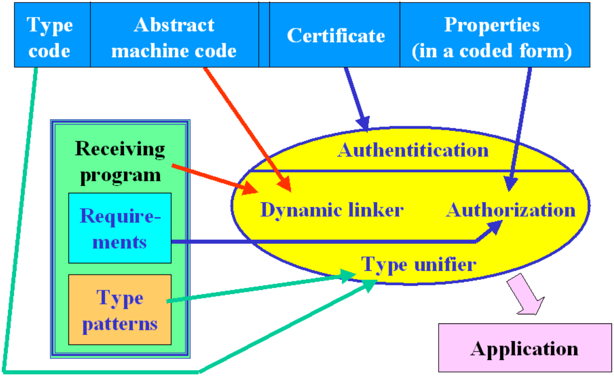
Producing verified mobile code



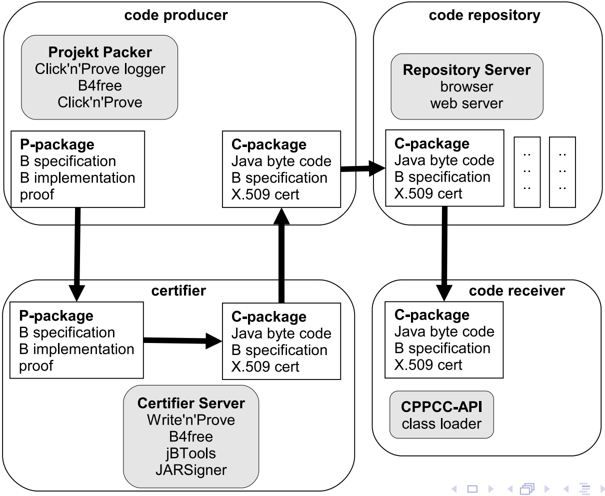
Certification of verified mobile code



Executing the verified mobile code



CPPCC: B-method and Java bytecode



Summary

- We have extended an existing proof tool for Clean with support for temporal properties and designed the proof tactics necessary to manipulate them.
- Subtype marks provide a way to annotate types with invariants, and establish a co-operation between a type checker and a proof system.
- Certified Proved Property Carrying Code framework: efficient verification of the correctness of mobile components.

Related projects

- Expressing and proving temporal properties of Clean programs
- Annotations for expressing subtype invariants
- Design of Distributed Clean
- Safe transformations: refactoring (Clean, Erlang)
- Safe destructive update of data structures