Behavior models and verification

Lecture 13

Jan Kofroň, František Plášil
Efficient verification algorithms can extend applicability of formal methods

Many systems can be decomposed into parts
- verifying each properties of parts separately
- if conjunction of parts properties implies overall specification, we are done
- the entire system never analyzed as whole
Communication protocol: Sender, network, receiver

Overall specification:
- Data correctly transmitted from sender to receiver

Partial specifications
- Data correctly sent from sender to network
- Data correctly transmitted via network
- Data correctly transmitted from network to receiver

Verification of partial specifications typically much easier
- sum of the state spaces much smaller than state space of entire system (recall state space explosion)
Assume-guarantee for composition reasoning

- Verifies each component separately
- Based on specification of
  - **Assumptions** – requirements on behavior of environment
  - **Guarantees** – provisions offered to environment if assumptions are met
  - environment = the other components

- By combining assumptions and guarantees of particular parts, it is possible to establish correctness of entire system
  - Full transition graph never constructed
Formally

- Formula is triple $\langle g \rangle M \langle f \rangle$ where $g$ and $f$ are temporal formulae and $M$ is a program
  - true if whenever $M$ is part of system satisfying $g$ system also guarantees $f$

- Typically: proof for $\langle g \rangle M' \langle f \rangle$ and $\langle true \rangle M \langle g \rangle$
  - then we have $\langle true \rangle M \parallel M' \langle f \rangle$

- Can be expressed as inference rule:

$$
\frac{
\langle true \rangle M \langle g \rangle \\
\langle g \rangle M' \langle f \rangle
}{
\langle true \rangle M \parallel M' \langle f \rangle
}$$

Formally

- Necessary to avoid circular dependencies:
  \[
  \langle f \rangle M \langle g \rangle \\
  \langle g \rangle M' \langle f \rangle \\
  \hline \\
  M \parallel M' \models f \land g
  \]

- This is unsound!
- Should be avoided
Model level
Applications – SW components

• Each component specifies not only provided (implemented) interfaces,
  ▪ as objects do
• But also required ones
  ▪ in addition to objects

• Syntactic (type) information
  ▪ may or may not consider interface/type inheritance

• Semantic (behavior) specification
  ▪ verification techniques, usually model checking or equivalence checking (simulation, bisimulation, ...)

Jan Kofroň, František Plášil, Lecture 13
In hierarchical component models (SOFA, Fractal)

- i.e., components are nested (primitive vs. composite)
- two directions of composability
  - horizontal – composition of subcomponents of a component
  - vertical – compatibility of subcomponents of a component and the component itself

Architecture description in form of

- IDL/ADL
- (UML/EMF) model of application
- usually includes many aspects (static – connections, dynamic – behavior, extra-functional – performance, ...)
Type information

- Assumptions – requirements on types of required interfaces
- Provisions – types of provided interfaces
- Trivial to check composability
  - even if type hierarchy (inheritance) is considered
  - both horizontal and vertical directions straightforward
Can cover various aspects of the components
  • functional – sequences of messages/calls
  • extra-functional
    • timing
    • reliability
    • resource usage
    • security
    • ...

Composition verification based on the same principle
  • component should provide at least as much (as good, fast, reliable, ...) as its counterpart requires
Functional aspects

- At provided side – specification of allowed sequences of method calls/messages
  - interfaces compatible if subset of provided is required
- At required part similarly
  - in reaction to provided ones
  - spontaneous – internal threads

- Can be extended to entire component
  - e.g. frame protocol in SOFA
  - specifies behavior of entire component
  - simplifies reuse and automatic selection of components
  - characterize component semantics at abstract level
Many options to define composition
- language subset
  - with/without restriction to common methods
  - ...
- LTS-based comparison
  - simulation preorder
  - (weak) bisimulation
  - ...

Depends on properties we want to preserve
- and on what components are prepared for
Extra-functional aspects

• Similar in many views to functional case
  ▪ “provide at least what is required”

• Granularity from method/service/function to component/set of components

• Can cover various dimensions
  ▪ performance – execution time (average/worst)
  ▪ resources – required CPU/memory amount
  ▪ reliability – likelihood of failure, time to recovery
  ▪ specification of HW for deployment
  ▪ …
Extra-functional aspects – Example

• Palladio Component Model (PCM)
  ▪ required resources
  ▪ performance
  ▪ deployment
  ▪ usage profile

• Many types of analysis
  ▪ worst/mean execution time for composed services
  ▪ cumulated resource requirements
  ▪ possible deployment targets for given usage profile
  ▪ reliability analysis

• See: http://www.palladio-simulator.com/
Code level
Code level

- Types – usually checked by compiler
  - no additional effort required

- Semantics – code contracts
  - at level of functions/methods
  - assumptions – preconditions
  - guarantees – postconditions
  - usually also invariants
    - helps with verification
    - loop invariants
Verification is then **modular**

- each function is verified separately – whether the code really guarantees postcondition once precondition is satisfied at function entry
- if function is called from within other function, its contract is assumed (precondition is checker, postcondition is assumed)
public class ArrayList {
    public void add(int index, Object obj) { ... }
    public int size() { ... }
}

• “Value of the index parameter has to be greater than or equal to zero. Successful call of add increases the size of the array by one.”

• Formally:

    public void add(int index, Object obj) {
        requires index >= 0;
        ensures size = old(size) + 1;
    }
Code level – Approaches

- Contract specification languages
  - Spec#, JML, Code Contracts, ...

- There are tools to verify contracts
  - model checkers, SAT/SMT solvers, theorem provers

- NSWI132 – Program analysis and code verification
It is not easy to specify the contracts
- preconditions
  - too weak to guarantee postconditions
  - too strong to be satisfied by caller
- postconditions
  - too strong to be proven
  - too weak to “satisfy” caller

One has to know and tune...
int[N] field;
int swapMin(int from)
{
    \textit{swaps the min value beyond from with the one at from and return the index}
}

int main()
{
    // sorted
    \textbf{ensures} (\textbf{forall} \textbf{int} \textbf{i} : 0 < i < N-2 : \textbf{field}[i] \leq \textbf{field}[i+1]);
    // the original values
    \textbf{ensures} (\textbf{forall} \textbf{int} \textbf{i} : 0 < i < N-1 \quad \&\& \quad \textbf{exists} \textbf{int} \textbf{j} : \text{\text{old}}(\textbf{field}[\textbf{j}]) = \textbf{field}[\textbf{i}]);

    \{ 
        \textbf{for} (\textbf{int} \textbf{i} = 0; \textbf{i} < \textbf{N}; \textbf{i}++) 
        \quad \textbf{swapMin}(\textbf{i});
    \}
}

We need some guarantees from swapMin to prove this
int[N] field;

int swapMin(int from)
    ensures ((field[return] == old(field[from]) && old(field[from]) == field[return]));
{
    swaps the min value beyond from with the one at from and return the index
}

int main()
    // sorted
    ensures (forall int i : 0<i<N-2 : field[i] <= field[i+1]);
    // the original values
    ensures (forall int i : 0<i<N-1 && exists int j : old(field[j]) == field[i]);
{
    for (int i = 0; i < N; i++)
        swapMin(i);
}
int[N] field;
int swapMin(int from)
  ensures (forall int i<from: field[i]<=field[from]);
{
  swaps the min value beyond from with the one at from and return the index
}

int main()
  // sorted
  ensures (forall int i : 0<i<N-2 : field[i] <= field[i+1]);
  // the original values
  ensures (forall int i : 0<i<N-1 && exists int j : old(field[j]) == field[i]);
{
  for (int i = 0; i < N; i++)
    swapMin(i);
}
Code-to-Model Compliance
Based on checking model level specification with code
  - whether code complies to model spec
  - again modular – at granularity of
    - functions/methods
    - objects
    - sw components
  - similar to code contracts but usually coarser granularity
    - e.g., limited to sequences of method calls

Allows for checking compositionality at model level
  - which is usually easier than at code level
  - handling components as annotated black boxes
  - if strong enough, entire problem **undecidable**
    - code model checking $\rightarrow$ halting problem
• Can employ functional and extra-functional aspects
  - call sequences
  - resource demands
  - worst-case execution time
  - reliability
Model level spec of DB component:
(f.open; (f.read {!fs.read} + f.write{!fs.write}))* ; f.close) | m.status*

Code:

```c
boolean open(FILE *file) { ... }

int read(int n, char *buffer) {
    assert(buffer);
    assert(n>0);
    return fs->fread(file, n, buffer);
}

void close(FILE *file) {
    assert(file.isopen());
    file.close();
}
```

...
Things to check:
- no assertion violated in any execution
  - i.e., adherence to specified provisions
- no required method called out of order
  - i.e., adherence to specified requirements

If verified, spec used in composition verification instead of code
- NAIL094
- Planned for Summer semester 2019/2020
- Inside into algorithms and techniques inside SAT (and SMT) solvers
NSWI132

Taught in Summer semester 2018/2019

Insight into code verification tools
- Focus on practical experience with the code verifiers