Discover Dezyne

The easiest way to build verifiably correct embedded software

Refinement in Dezyne
formal methods for the masses

Paul Hoogendijk, Verum
Dutch Model Checking Day 2018
Challenges for software

Solution: Reap the benefits of formal methods
Applying formal methods in industry

Common challenges:

1. Need to be expert in formal methods
   - How to model my system and requirements?
   - Does what I have modeled reflect my system/requirements?
   - How to interpret model checking result for my application?
   - ...

2. Non-scalable due to state explosion
   - Real world application are large (50K – 10M lines of code)
   - Many variables; large state space
Dezyne: formal methods for the masses

Solution to the common challenges:

1. **Two level** approach:
   - **Dezyne language** relates to *common software engineers*
     - State machine + imperative language
   - Model checker hidden for user
   - **Dezyne language translated to mCLR2 language**
   - Counter example translated *back as sequence diagram in Dezyne*
   - Generate **executable code from Dezyne code**

2. **Compositional** solution
   - Component based: **interfaces + components**
   - Interfaces have behaviour (!)
   - Component and its requires interfaces **refine** provides interfaces
Two level approach

- dezyne model
  - Generate formal model
  - Generate source code

- model checking
  - equivalent behaviours

- executable code

Design Errors
Where is our tooling used?
Where is our tooling used?

Multiple projects

750K lines of code

500K lines of code
DEMO
Refinement in Dezyne

Dezyne provides interface compliance

≡

Refinement between component and provides interface (restricted to alphabet of provides interface)

Some compliance examples in Dezyne:
Interface compliance examples:

```java
interface I {
    in bool e();
    behaviour {
        on e: reply(false);
        on e: reply(true);
    }
}
```

UI Interface I is correctly implemented by component C:

```java
component C {
    provides I p;
    behaviour {
        on p.e(): reply(true);
    }
}
```
Interface compliance examples:

interface I {
    in void e();
    behaviour {
        on e: {}
        on f: {}
    }
}

component C {
    provides I p;
    behaviour {
        on p.e(): {}
    }
}
Interface compliance examples:

```java
interface I {
    in void e();
    behaviour {
        on e: {} 
        on f: {} 
    }
}
```

Interface I is incorrectly implemented by component C:

```java
component C {
    provides I p;
    behaviour {
        on p.e(): {}
        on p.f(): illegal
    }
}
```

Component is made complete: non handled events are regarded as illegal.
Interface compliance examples:

interface I {
    in bool e();
    behaviour {
        on e: reply(false);
        on e: reply(true);
    }
}

Interface I is correctly implemented by component C:

component C {
    provides I p;
    requires I r;
    behaviour {
        on p.e(): reply(!r.e());
    }
}
Interface compliance examples:

```java
interface I {
    in bool e();
    behaviour {
        on e: reply(false);
    }
}
```

Interface I is incorrectly implemented by component C:

```java
component C {
    provides I p;
    requires I r;
    behaviour {
        on p.e(): reply(!r.e());
    }
}
```
Interface compliance examples:

```plaintext
interface I {
    out void cb();
    behaviour {
        on inevitable: cb;
    }
}

component C {
    provides I p;
    requires I r;
    behaviour {
        on r.cb(): p.cb();
    }
}
```

Interface I is correctly implemented by component C:
Interface compliance examples:

```java
interface I {
    out void cb();
    behaviour {
        on inevitable: cb;
    }
}
```

Interface I is incorrectly implemented by component C:

```java
component C {
    provides I p;
    requires I r;
    behaviour {
        on r.cb(): {}
    }
}
```
Interface compliance examples:

```java
interface I {
    out void cb();
    behaviour {
        on optional: cb;
    }
}
```

Interface I is correctly implemented by component C:

```java
component C {
    provides I p;
    requires I r;
    behaviour {
        on r.cb(): {}
    }
}
```
Verification backend

- Previously FDR used in verification backend
- Started developing with mCLR2 end of 2014
  - Tetracom project between Verum and TU/e
  - mCRL2 replaced FDR as of release 2.7.0 (march 2018)
- FDR vs mCRL2:
  - FDR: Failures-Divergences Refinement
    - $\text{Impl} \leq \text{Spec} \equiv \text{failures}(\text{Impl}) \subseteq \text{failures}(\text{Spec})$
    - $\text{failures}(P) = \{ (\text{tr}, X) | \text{tr} \in \text{traces}(P), X \in \text{refusals}(P \text{ after } \text{tr}) \}$
  - FDR each assert expressed as refinement property
  - FDR cannot handle fairness
    - Using FDR for functional verification results in many livelocks which hides refinement issue 😞
- mCLR2 does handle fairness 😊
Verification flow in mCRL2

```
cat hello.dzn
| parse          dzn -> ast
| codegen-mcrl2  ast -> mcrl2
| mcrl22lps      mcrl2 -> lps (linear proc. spec)
| lps2lts        lps -> lts
| ltsconvert     lts -> lts (reduction)
| lts-check      lts -> lts (add refusals+check)

> hello.lts
```

```
ltscompare -pweak-failures hello.lts intf.lts
```
Verification flow in mCRL2

```
  cat hello.dzn
  | parse               dzn -> ast
  | codegen-mcrl2       ast -> mcrl2
  | mcrl22lps           mcrl2 -> lps (linear proc. spec)
  | lps2lst             lps -> lts
  | ltsconvert          lts -> lts (reduction)
  | lts-check           lts -> lts (add refusals+check)
  > hello.lts

  ltscompare -pweak-failures hello.lts intf.lts
```

mCRL2 tooling from TU/e, Jan Friso Groote e.a.
Verification flow in mCRL2

cat hello.dzn
| parse
| codegen-mcrl2
| mcrl22lps
| lps2lts
| ltsconvert
| lts-check
> hello.lts

dzn -> ast
ast -> mcrl2
mcrl2 -> lps (linear proc. spec)
lps -> lts (reduction)
lts -> lts (add refusals+check)

Check on LTS:
• Non-determinism
• Illegal
• Deadlock
• Livelock

Late introduction of refusals for optional events

mCRL2 tooling from TU/e, Jan Friso Groote e.a.

Failures Refinement between component and requires interfaces and provides interfaces
Compositionality due to refinement

Model checker proves:
- $I_1 \parallel C_0 \leq I_0$, $I_2 \parallel C_1 \leq I_1$, $C_2 \leq I_2$
- $C_0, C_1, C_2$ free of deadlock, livelock, illegal, and deterministic

From which we conclude
- $C_0 \parallel C_1 \parallel C_2 \leq I_0$ due to monotonicity of $\parallel$ w.r.t. failures refinement
- $C_0 \parallel C_1 \parallel C_2$ free of livelock, illegal, and deterministic (due to traces), and deadlock (due to refusals)
Consistency verification & generated code

For each supported language:
For each component of test set:
- Code is generated plus test-stub
- Set of traces covering the component lts is generated
- Each trace is replayed on test executable of component:
  - All in events are fed to test-stub around component
  - Both in and out events are logged by stub:
    - trace log of component needs to be the same as original trace
interface async {
    in void doit();
    out void done();
    behaviour {
        bool idle = true;
        [idle] on doit: idle=false;
        ![idle] {
            on inevitable: { done; idle=true;}
        }
    }
}
Optional/inevitable: asynchronous events

```java
interface async {
    in void doit();
    out void done();
    behaviour {
        bool idle = true;
        [idle] on doit: idle=false;
        ![idle] {
            on inevitable: { done; idle=true; }
        }
    }
    event “inevitable” relates to internal event of underlying component, hence, is hidden.
}
```
Optional/inevitable: asynchronous events

```
interface async {
    in void doit();
    out void done();
    behaviour {
        bool idle = true;
        [idle] on doit: idle=false;
        ![idle] {
            on optional: { done; idle=true;}
        }
    }
}
```
interface async {
    in void doit();
    out void done();
    behaviour {
        bool idle = true;
        [idle] on doit: idle=false;
        ![idle] {
            on optional: { done; idle=true;}
        }
    }
}

Event “optional” may be refused, hence, this interface deadlocks
Inevitable/optional: translation in mCRL2

on inevitable: callback;
on e: {}

versus

on optional: callback;
on e: {}

\[ P = \text{inevitable} \rightarrow \text{callback} \rightarrow P \]
\[ | \ e \rightarrow \text{return} \rightarrow P \]

\[ P = \text{optional} \rightarrow \text{callback} \rightarrow P \]
\[ | \ e \rightarrow \text{return} \rightarrow P \]
\[ | \ tau \rightarrow P' \]
\[ P' = e \rightarrow \text{return} \rightarrow P \]
Inevitable/optional: translation in mCRL2

on inevitable: callback;
on e: {}

versus
on optional: callback;
on e: {}

P = inevitable -> callback -> P
| e -> return -> P

P = optional -> callback -> P
| e -> return -> P
| tau -> P'
P' = e -> return -> P

tau transition to copy of state where “optional” is removed.
Hence, event “optional” can be refused in state P
Late introduction of refusals

- Having many “optionals” in requires interfaces leads to state explosion during lts generation:
  - \( \text{mcrl22lts(} \)
    \[
    \text{mcrl2(}C) \]
    \[
    \text{|| mcrl2-plus-refusals(I0) x2} \]
    \[
    \text{|| mcrl2-plus-refusals(I1) x2} \]
    \[
    \text{|| mcrl2-plus-refusals(I2) x2 = x8} \]
    \(
    ) \text{ where mcrl2, mcrl2-plus-refusals: dzn} \rightarrow \text{mcrl2}
  
- Solution:
  - Add refusals, i.e. duplicated states, as late as possible:
    \[
    \text{add-refusals(ltsconvert(} \)
    \[
    \text{mcrl22lts(mcrl2(}C) || mcrl2(I0) || mcrl2(I1) || mcrl2(I2) \)
    \]
    \[
    \text{)) where add-refusals: lts} \rightarrow \text{lts} \]
    thus, just before deadlock and compliance check, and after lts reduction by ltsconvert
Late introduction of refusals

- Inspired by how FDR internally works:
  - FDR constructs GLTS i.s.o. LTS: (G=Generalized)
    - GLTS, amongst others:
      - LTS plus for each node, maximum refusal set.
    - Whether event can be refused or not, does not increase size of GLTS (!)

- Reduced verification time back from several minutes to few seconds for some of our customer models.
  - Now comparable to FDR based verification time
Conclusion

- Dezyne allows regular software engineers to construct industrial size software systems while reaping the power of formal methods.
  - Two level approach,
  - Compositionality (due to use of failures refinement)
- Introducing mCRL2 has been an pleasant and inspiring journey
  - Very pleasant cooperation with TU/e, real win/win.
  - Using new back-end caused no visible change for users
  - Performance is on-par, sometimes faster, than FDR
    - Late introduction of refusals was essential in this.
  - Enables extension towards functional & system verification
Thank You

Acknowledgments:

- mCRL2 team of TU/e:
  - Jan Friso Groote
  - Tim Willemse
  - Wieger Wesselink
- Verum team

Questions?