Proving Necessity

Designing a Proof System for Necessary Conditions

Julian Mackay, James Noble, Sophia Drossopolou, Susan Eisenbach

Traditional Specifications

Traditional Specifications: Correctness

- {P} s {Q}

Traditional Specifications: Correctness

- {P} s {Q}
- if P is true before we execute s, then Q will be true after

Traditional Specifications: Correctness

```
class Account
  field balance
  field password
  method transfer(pwd, to, amt)
    if this.auth(pwd)
      this.balance -= amt
      to.balance += amt
  method send(amt)
    this.balance += amt
```

- {P} s {Q}
- if P is true before we execute s, then Q will be true after
- e.g.

Traditional Specifications: Correctness

- {P} s {Q}
- if P is true before we execute s, then Q will be true after
- e.g.

```
class Account
  field balance
  field password
  method transfer(pwd, to, amt)
    if this.auth(pwd)
      this.balance -= amt
      to.balance += amt
  method send(amt)
    this.balance += amt
```

```
{pwd = a.password}a.transfer(pwd,to,amt){a.balance_new = a.balance_old - amt}
```

Necessity Specifications

- open world

```
class Account
  field balance
  field password
  method transfer(pwd, to, amt)
    if this.auth(pwd)
      this.balance -= amt
      to.balance += amt
  method send(amt)
    this.balance += amt
  …
```

Necessity Specifications

- open world

```
class Account
  field balance
  field password
  method transfer(pwd, to, amt)
    if this.auth(pwd)
      this.balance -= amt
      to.balance += amt
  method send(amt)
    this.balance += amt
  …
```

Necessity Specifications

- open world
- what are the necessary conditions for our balance to decrease?

Necessity Specifications

```
class Account
  field balance
  field password
  method transfer(pwd, to, amt)
    if this.auth(pwd)
      this.balance -= amt
      to.balance += amt
  method send(amt)
    this.balance += amt
  …
```

- open world
- what are the necessary conditions for our balance to decrease?

if the balance decreases then

```
class Account
  field balance
  field password
  method transfer(pwd, to, amt)
    if this.auth(pwd)
      this.balance -= amt
      to.balance += amt
  method send(amt)
    this.balance += amt
  …
```

Necessity Specifications

- open world
- what are the necessary conditions for our balance to decrease?

if the balance decreases then

- the password was correct?

```
class Account
  field balance
  field password
  method transfer(pwd, to, amt)
    if this.auth(pwd)
      this.balance -= amt
      to.balance += amt
  method send(amt)
    this.balance += amt
  …
```

Necessity Specifications

- open world
- what are the necessary conditions for our balance to decrease?

if the balance decreases then

- the password was correct?
- transfer was called? i.e. is there some other way to drain our account?

```
class Account
  field balance
  field password
  method transfer(pwd, to, amt)
    if this.auth(pwd)
      this.balance -= amt
      to.balance += amt
  method send(amt)
    this.balance += amt
  …
```

Necessity Specifications

- open world
- what are the necessary conditions for our balance to decrease?

if the balance decreases then

- the password was correct?
- transfer was called? i.e. is there some other way to drain our account?
- someone knew our password?

Necessity Specifications

```
class Account
  field balance
  field password
  method transfer(pwd, to, amt)
    if this.auth(pwd)
      this.balance -= amt
      to.balance += amt
  method send(amt)
    this.balance += amt
  …
```

- open world
- what are the necessary conditions for our balance to decrease?

if the balance decreases then

- the password was correct?
- transfer was called? i.e. is there some other way to drain our account?
- someone knew our password?

what if we wanted to specify the behaviour of arbitrary code?

Necessity Specifications

$$\texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A$$

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A$$

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A$$

Necessity Specifications

$$\texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A$$

if A1 is true, and A2 is true in the next program state, then A must have originally been true too

- e.g. if a.balance = 100 and in the next visible program state a.balance < 100, then transfer have been called

Necessity Specifications

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A$$

if A1 is true, and A2 is true in some future program state, then A must have originally been true too

- e.g. if a.balance = 100 and in some future program state a.balance < 100, then someone must know our password

Necessity Specifications

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A$$

if A1 is true, and A2 is true in some future program state, then A must have been true in some intermediate state

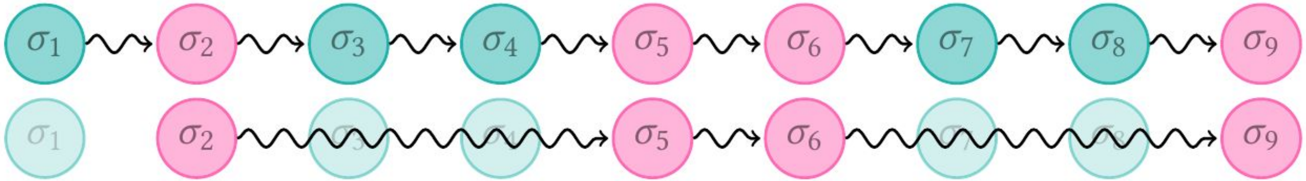
- e.g. if a.balance = 100 and in some future program state a.balance < 100, then someone have called transfer with the correct password in some intermediate program state

Necessity Specifications - Open World

- simple module system, i.e. just a set of classes
- internal vs external
- necessity specs describe the externally visible world, i.e. externally visible program states

Necessity Specifications - Modelling the Open World

- simple module system, i.e. just a set of classes
- internal vs external
- necessity specs describe the externally visible world, i.e. externally visible program states



Necessity Specifications

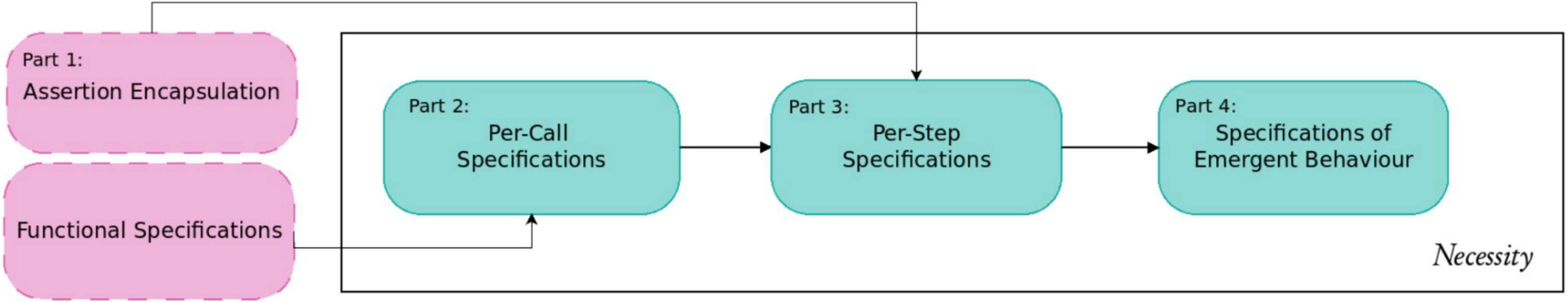
- necessity specs describe the externally visible world, i.e. externally visible program states

`from` A_1 `to` A_2 `onlyIf` A:

`from` A_1 `next` A_2 `onlyIf` A:

`from` A_1 `to` A_2 `onlyThrough` A:

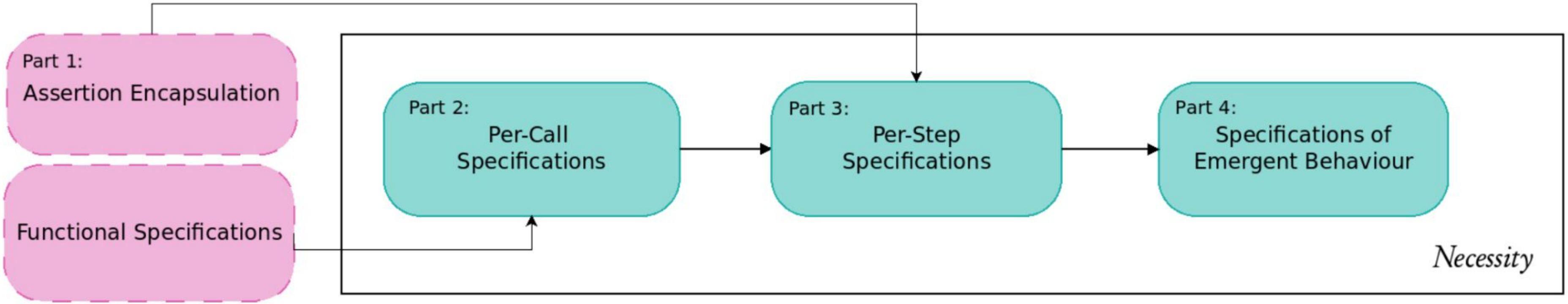
Necessity Specifications - How we reason about necessity



Necessity Specifications - How we reason about necessity

we base our logic on top of two secondary proof systems

- assertion encapsulation: what assertions require internal library code to be invalidated
- functional specifications: proofs of traditional Hoare triples



Necessity Specifications - How we reason about necessity

our logic consists of 3 components:

1. per-call necessary preconditions
   - e.g. for a call to transfer to decrease the balance, the correct password must be given

```
from a.balance = bal && <_ calls a.transfer(pwd, _, _)>
  next a.balance < bal
  onlyIf pwd = a.password
```

we build this off traditional specifications ….

Necessity Specifications - How we reason about necessity

```
from a.balance = bal && <_ calls a.transfer(pwd, _, _)>
  next a.balance < bal
  onlyIf pwd = a.password
```

equivalent to …

```
{a.balance = bal && pwd <> a.password}
  a.transfer(pwd, _, _)
{a.balance = bal}
```

Necessity Specifications - How we reason about necessity

our logic consists of 3 components:

2. per-step necessary preconditions
   - e.g. for any arbitrary execution step to decrease the balance, that step must be a call to transfer with the correct password

```
from a.balance = bal
  next a.balance < bal
  onlyIf <_ calls a.transfer(pwd, _, _)> && pwd = a.password
```

we build this on top of assertion encapsulation …

Necessity Specifications - How we reason about necessity

```
from a.balance = bal
  next a.balance < bal
  onlyIf <_ calls a.transfer(pwd, _, _)> && pwd = a.password
```

is true iff

- `a : Account`
- `a.balance = bal is "encapsulated", i.e. requires internal computation to invalidate`
- `there is no other way to decrease the balance internally`

Necessity Specifications - How we reason about necessity

our logic consists of 3 components:

3. emergent behaviour
    - e.g. for any execution of arbitrary length to decrease the balance, someone must first know the password

```
from a.balance = bal
  to a.balance < bal
  onlyIf exists x.[<x access a.password> && <external x>]
```

Necessity Specifications - How we reason about necessity

our logic consists of 3 components:

3. emergent behaviour
    - e.g. for any execution of arbitrary length to decrease the balance, someone must first know the password

```
from a.balance = bal
  to a.balance < bal
  onlyIf exists x.[<x access a.password> && <external x>]
```

note, the above spec does not specify any functions in the module interface, it only specifies a relationship between two fields. In fact, with ghost fields we are able to remove all mention of implementation details altogether

Necessity Specifications - How we reason about necessity

our logic consists of 3 components:

3. emergent behaviour
    - e.g. for any execution of arbitrary length to decrease the balance, someone must first know the password

```
from a.balance() = bal
  to a.balance() < bal
  onlyIf exists x.[<x access a.password> && <external x>]
```

note, the above spec does not specify any functions in the module interface, it only specifies a relationship between two fields. In fact, with ghost fields we are able to remove all mention of implementation details altogether

Necessity Specifications - How we reason about necessity … some math stuff

$$\frac{M \vdash \{x : C \land P_1 \land \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\neg P_2\}}{M \vdash \texttt{from}\ P_1 \land x : C \land \langle \_\ \texttt{calls}\ x.m(\overline{z}) \rangle\ \texttt{next}\ P_2\ \texttt{onlyIf}\ P} \quad \text{(If1-Classical)}$$

$$\frac{M \vdash \{x : C \land \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\texttt{res} \neq y\}}{M \vdash \texttt{from}\ \texttt{inside}(y) \land x : C \land \langle \_\ \texttt{calls}\ x.m(\overline{z}) \rangle\ \texttt{next}\ \neg\texttt{inside}(y)\ \texttt{onlyIf}\ P} \quad \text{(If1-Inside)}$$

Necessity Specifications - How we reason about necessity … some math stuff

$$\frac{\begin{array}{c}\left[\begin{array}{c} \textit{for all } C \in dom(M) \textit{ and } m \in M(C).\texttt{mths}, \\ [M \vdash \texttt{from}\, A_1 \wedge x : C \wedge \langle \_ \,\texttt{calls}\, x.m(\overline{z})\rangle\, \texttt{next}\, A_2\, \texttt{onlyIf}\, A_3] \end{array}\right] \\ M \vdash A_1 \longrightarrow \neg A_2 \qquad M \vdash A_1 \Rightarrow Enc(A_2) \end{array}}{M \vdash \texttt{from}\, A_1\, \texttt{next}\, A_2\, \texttt{onlyIf}\, A_3} \quad (\textsc{If1-Internal})$$

$$\frac{M \vdash A_1 \longrightarrow A_1' \qquad M \vdash A_2 \longrightarrow A_2' \qquad M \vdash A_3' \longrightarrow A_3 \qquad M \vdash \texttt{from}\, A_1'\, \texttt{next}\, A_2'\, \texttt{onlyIf}\, A_3'}{M \vdash \texttt{from}\, A_1\, \texttt{next}\, A_2\, \texttt{onlyIf}\, A_3} \quad (\textsc{If1-}\longrightarrow)$$

$$\frac{M \vdash \texttt{from}\, A_1\, \texttt{next}\, A_2\, \texttt{onlyIf}\, A \vee A' \qquad M \vdash \texttt{from}\, A'\, \texttt{to}\, A_2\, \texttt{onlyThrough}\, \texttt{false}}{M \vdash \texttt{from}\, A_1\, \texttt{next}\, A_2\, \texttt{onlyIf}\, A} \quad (\textsc{If1-}\vee\textsc{E})$$

$$\frac{\forall y,\ M \vdash \texttt{from}\, ([y/x]A_1)\, \texttt{next}\, A_2\, \texttt{onlyIf}\, A}{M \vdash \texttt{from}\, \exists x.[A_1]\, \texttt{next}\, A_2\, \texttt{onlyIf}\, A} \quad (\textsc{If1-}\exists_1)$$

Necessity Specifications - How we reason about necessity … some math stuff

$$\frac{M \vdash \texttt{from}\ A\ \texttt{next}\ \neg A\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A\ \texttt{to}\ \neg A\ \texttt{onlyThrough}\ A'} \quad (\textsc{Changes})$$

$$\frac{\begin{array}{c} M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3 \\ M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_3\ \texttt{onlyThrough}\ A \end{array}}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A} \quad (\textsc{Trans}_1)$$

$$\frac{\begin{array}{c} M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3 \\ M \vdash \texttt{from}\ A_3\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A \end{array}}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A} \quad (\textsc{Trans}_2)$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A} \quad (\textsc{If})$$

$$M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_2 \quad (\textsc{End})$$

Necessity Specifications - How we reason about necessity … some math stuff

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3 \qquad M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_3\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A} \quad \text{(If-Trans)}$$

$$M \vdash \texttt{from}\ x : C\ \texttt{to}\ \neg x : C\ \texttt{onlyIf}\ \texttt{false} \quad \text{(If-Class)} \qquad M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A_1 \quad \text{(If-Start)}$$

Relationship with Temporal Logic

- These are proofs of temporal properties
- closely related to temporal logic, so how do they compare?

Relationship with Temporal Logic

- These are proofs of temporal properties
- closely related to temporal logic, so how do they compare?

$$\texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A$$

Relationship with Temporal Logic

- These are proofs of temporal properties
- closely related to temporal logic, so how do they compare?

$$\texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A$$

$$A_1 \ \land \ \bigcirc A_2 \ \longrightarrow \ A$$

Relationship with Temporal Logic

- These are proofs of temporal properties
- closely related to temporal logic, so how do they compare?

$$\texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A$$

$$A_1 \ \wedge \ \bigcirc A_2 \ \longrightarrow \ A$$

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A$$

Relationship with Temporal Logic

- These are proofs of temporal properties
- closely related to temporal logic, so how do they compare?

$$\texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A$$

$$A_1 \wedge \bigcirc A_2 \longrightarrow A$$

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A$$

$$A_1 \wedge \Diamond A_2 \longrightarrow A$$

Relationship with Temporal Logic

- These are proofs of temporal properties
- closely related to temporal logic, so how do they compare?

$$\texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A$$

$$A_1 \ \wedge\ \bigcirc A_2 \ \longrightarrow\ A$$

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A$$

$$A_1 \ \wedge\ \Diamond A_2 \ \longrightarrow\ A$$

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A$$

Relationship with Temporal Logic

- These are proofs of temporal properties
- closely related to temporal logic, so how do they compare?

$$\texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A$$

$$A_1 \ \wedge\ \bigcirc A_2 \ \longrightarrow\ A$$

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A$$

$$A_1 \ \wedge\ \Diamond A_2 \ \longrightarrow\ A$$

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A$$

???????????

Where to next?

- Modules are limited
  - If M1 is a dependency of M2, and satisfies our necessity spec, can we prove M1 also satisfies that necessity spec?
  - can we write specs about scopes, and not modules?

Where to next?

- Modules are limited
  - If M1 is a dependency of M2, and satisfies our necessity spec, can we prove M1 also satisfies that necessity spec?
  - can we write specs about scopes, and not modules?
- we can use traditional specs to prove necessity specs, can we prove traditional specs from necessity specs?
  - i.e. if we know a module observes some behaviour, can we use that in proving post-conditions to functions
  - `from true to ~ A onlyIf false => A`

Where to next?

- Modules are limited
  - If M1 is a dependency of M2, and satisfies our necessity spec, can we prove M1 also satisfies that necessity spec?
  - can we write specs about scopes, and not modules?
- we can use traditional specs to prove necessity specs, can we prove traditional specs from necessity specs?
  - i.e. if we know a module observes some behaviour, can we use that in proving post-conditions to functions
  - `from true to ~ A onlyIf false => A`
- currently we do not allow external calls from internal code
  - we are currently working on an extension that allows unrestricted external calls and re-entrant code

Where to next?

- Modules are limited
  - If M1 is a dependency of M2, and satisfies our necessity spec, can we prove M1 also satisfies that necessity spec?
  - can we write specs about scopes, and not modules?
- we can use traditional specs to prove necessity specs, can we prove traditional specs from necessity specs?
  - i.e. if we know a module observes some behaviour, can we use that in proving post-conditions to functions
  - `from true to ~ A onlyIf false => A`
- currently we do not allow external calls from internal code
  - we are currently working on an extension that allows unrestricted external calls and re-entrant code
- currently assertion encapsulation is constructed from an ad-hoc system relying on a rudimentary type system, can we define a full fledged proof system for encapsulation?