Deployment Components with Parametric Concurrency

Einar Broch Johnsen Olaf Owe
Rudolf Schlatte S. Lizeth Tapia Tarifa

University of Oslo

10 March 2011, Tallinn

HATS
Highly Adaptable and Trustworthy Software using Formal Models

SEVENTH FRAMEWORK PROGRAMME

http://www.hats-project.eu

Outline

1. Motivation and aim
2. The Abs language
3. Time model
4. Deployment components
5. Resource reallocation
6. Conclusions and future work

Motivation

Software systems tend to be released for a range of different architectures

Software systems tend to be released for a range of different architectures

Examples

- Software Product Lines
- Embedded Systems
- Sensors
- Web Services
- Operating Systems

Motivation

Software systems tend to be released for a range of different architectures

Examples

- Software Product Lines
- Embedded Systems
- Sensors
- Web Services
- Operating Systems

Need to model software which ranges over deployment scenarios

Abstraction levels in modeling

Specification level	Modeling formalisms
Design-oriented	Graphical notations
↕	↕
?	?
↕	↕
Implementation-oriented	Spec#, Java+JML

Motivation

Abstraction levels in modeling

Specification level	Modeling formalisms
Design-oriented	Graphical notations
↕	↕
Abstract behavioral	Abs
↕	↕
Implementation-oriented	Spec#, Java+JML

Abstract behavioral modeling language for distributed active objects

Abstract behavioral modeling language for distributed active objects

Syntactic categories.

C, I, m in Names
g in Guard
s in Stmt
x in Var
e in Expr
b in BoolExpr

Definitions.

$$
\begin{array}{rcl}
IF & ::= & \mathbf{interface}\ I\ \{\ [\overline{Sg}]\ \} \\
CL & ::= & \mathbf{class}\ C\ [(\overline{I\ x})]\ [\mathbf{implements}\ \overline{I}]\ \{\ [\overline{I\ x;}]\ \overline{M}\} \\
Sg & ::= & I\ m\ ([\overline{I\ x}]) \\
M & ::= & Sg == [\overline{I\ x;}]\ \{\ s\ \} \\
g & ::= & b \mid x? \mid g \wedge g \\
s & ::= & s; s \mid x := rhs \mid \mathbf{release} \mid \mathbf{await}\ g \mid \mathbf{return}\ e \\
 & \mid & \mathbf{if}\ b\ \mathbf{then}\ \{\ s\ \}\ [\mathbf{else}\ \{\ s\ \}] \mid \mathbf{while}\ b\ \{\ s\ \} \mid \mathbf{skip} \\
e & ::= & x \mid b \mid \mathbf{this} \mid\mid \mathbf{null} \\
rhs & ::= & e \mid \mathbf{new}\ C\ (\overline{e}) \mid [e]!m(\overline{e}) \mid [e.]m(\overline{e}) \mid x.\mathbf{get}
\end{array}
$$

Abs Modeling Language

Abstract behavioral modeling language for distributed active objects

Syntactic categories.

C, I, m in Names
g in Guard
s in Stmt
x in Var
e in Expr
b in BoolExpr

Definitions.

$$
\begin{array}{rcl}
IF & ::= & \texttt{interface}\ I\ \{\ [\overline{Sg}]\ \} \\
CL & ::= & \texttt{class}\ C\ [(\overline{I\ x})]\ [\texttt{implements}\ \overline{I}]\ \{\ [\overline{I\ x};]\ \overline{M}\} \\
Sg & ::= & I\ m\ ([\overline{I\ x}]) \\
M & ::= & Sg == [\overline{I\ x};]\ \{\ s\ \} \\
g & ::= & b \mid x? \mid g \wedge g \\
s & ::= & s;s \mid x := rhs \mid \texttt{release} \mid \texttt{await}\ g \mid \texttt{return}\ e \\
 & \mid & \texttt{if}\ b\ \texttt{then}\ \{\ s\ \}\ [\texttt{else}\ \{\ s\ \}] \mid \texttt{while}\ b\ \{\ s\ \} \mid \texttt{skip} \\
e & ::= & x \mid b \mid \texttt{this} \mid\mid \texttt{null} \\
rhs & ::= & e \mid \texttt{new}\ C\ (\overline{e}) \mid [e]!m(\overline{e}) \mid [e.]m(\overline{e}) \mid x.\texttt{get}
\end{array}
$$

- Abs has a model of parallelism based on concurrent objects, where the communications is through asynchronous method calls.
- Every object has a set of processes to be executed
- At most one process per object is *active*, the others are *suspended*

Abs Modeling Language

Abstract behavioral modeling language for distributed active objects

Syntactic categories.

C, I, m in Names
g in Guard
s in Stmt
x in Var
e in Expr
b in BoolExpr

Definitions.

$$
\begin{array}{rcl}
IF & ::= & \mathbf{interface}\ I\ \{\ [\overline{Sg}]\ \} \\
CL & ::= & \mathbf{class}\ C\ [(\overline{I\ x})]\ [\mathbf{implements}\ \overline{I}]\ \{\ [\overline{I\ x};]\ \overline{M}\} \\
Sg & ::= & I\ m\ ([\overline{I\ x}]) \\
M & ::= & Sg == [\overline{I\ x};]\ \{\ s\ \} \\
g & ::= & b \mid x? \mid g \wedge g \\
s & ::= & s; s \mid x := rhs \mid \mathbf{release} \mid \mathbf{await}\ g \mid \mathbf{return}\ e \\
 & \mid & \mathbf{if}\ b\ \mathbf{then}\ \{\ s\ \}\ [\mathbf{else}\ \{\ s\ \}] \mid \mathbf{while}\ b\ \{\ s\ \} \mid \mathbf{skip} \\
e & ::= & x \mid b \mid \mathbf{this} \mid\mid \mathbf{null} \\
rhs & ::= & e \mid \mathbf{new}\ C\ (\overline{e}) \mid [e]!m(\overline{e}) \mid [e.]m(\overline{e}) \mid x.\mathbf{get}
\end{array}
$$

- Abs has a model of parallelism based on concurrent objects, where the communications is through asynchronous method calls.
- Every object has a set of processes to be executed
- At most one process per object is *active*, the others are *suspended*
- Scheduling is controlled by await statements
- Compositional proof theory, implemented in KeY

A Time Model for Abs

- A **time interval** captures the execution between two observable points in time

A Time Model for Abs

- A **time interval** captures the execution between two observable points in time
- Comparable to a system clock which updates every n milliseconds

A Time Model for Abs

- A **time interval** captures the execution between two observable points in time
- Comparable to a system clock which updates every n milliseconds
- The expression **`now`** returns the present time

A Time Model for Abs

- A **time interval** captures the execution between two observable points in time
- Comparable to a system clock which updates every n milliseconds
- The expression **`now`** returns the present time
- Suitable for guards in **`await`** statements.

A Time Model for Abs

- A **time interval** captures the execution between two observable points in time
- Comparable to a system clock which updates every *n* milliseconds
- The expression **now** returns the present time
- Suitable for guards in **await** statements.
  - **Example:**

```
Time t:=now;
await now ≥ t + c;
```

A Time Model for Abs

- A **time interval** captures the execution between two observable points in time
- Comparable to a system clock which updates every n milliseconds
- The expression **now** returns the present time
- Suitable for guards in **await** statements.
  - **Example:**

```
Time t:=now;
await now ≥ t + c;
```

- **From the local perspective** time advances by
  - awaiting the passage of time, or
  - when no other activity may occur

A Time Model for Abs

- A **time interval** captures the execution between two observable points in time
- Comparable to a system clock which updates every *n* milliseconds
- The expression **now** returns the present time
- Suitable for guards in **await** statements.
  - **Example:**

```
Time t:=now;
await now ≥ t + c;
```

- **From the local perspective** time advances by
  - awaiting the passage of time, or
  - when no other activity may occur

Paper: **Lightweight Time Modeling in Timed Creol**

Proc. 1st Int. Workshop on Rewriting Techniques for Real-Time Systems (RTRTS 2010), ENTCS 36:67–81, 2010

Aim

Apply performance analysis to OO models which range over deployment scenarios

Aim

Apply performance analysis to OO models which range over deployment scenarios

Modeling of deployment scenarios

Aim

Apply performance analysis to OO models which range over deployment scenarios

Modeling of deployment scenarios

- *Deployment components* with a set of (physical) processors
- Every component is parametric in the amount of concurrent processing *resources*

Aim

Apply performance analysis to OO models which range over deployment scenarios

Modeling of deployment scenarios

- *Deployment components* with a set of (physical) processors
- Every component is parametric in the amount of concurrent processing *resources*

Processing resources are:

Aim

Apply performance analysis to OO models which range over deployment scenarios

Modeling of deployment scenarios

- *Deployment components* with a set of (physical) processors
- Every component is parametric in the amount of concurrent processing *resources*

Processing resources are:

- Shared between the concurrent objects of a deployment component
- Updated for every *time interval*

Approach

- Propose an abstract model of *deployment components*
  - Concurrent object groups
  - Parametric amount of resources per time interval

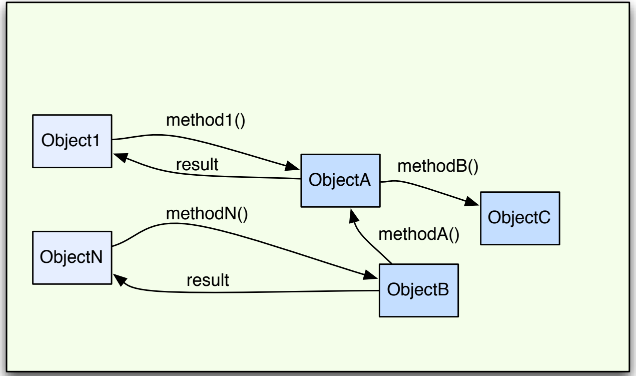
Approach

- Propose an abstract model of *deployment components*
  - Concurrent object groups
  - Parametric amount of resources per time interval
- Extend the *Abs* modeling language
  - Time model
  - Deployment components
  - Resource reallocation

Approach

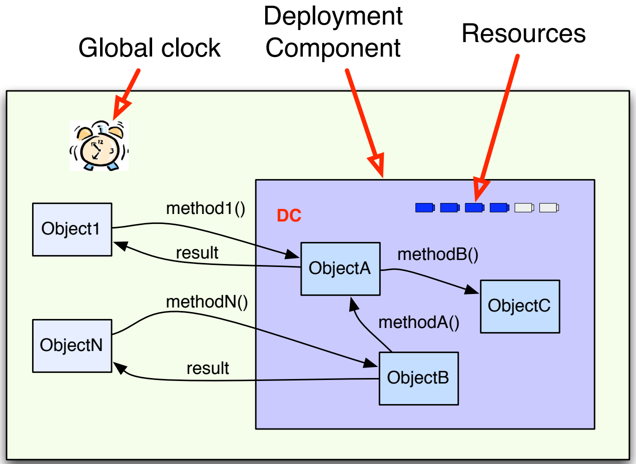
- Propose an abstract model of *deployment components*
  - Concurrent object groups
  - Parametric amount of resources per time interval
- Extend the *Abs* modeling language
  - Time model
  - Deployment components
  - Resource reallocation
- Operational semantics in rewriting logic
  - Executable prototype using Maude
  - Language interpreter
  - Simulation of model behavior
  - Test suites

Model



Object1
method1()
result
ObjectA
methodB()
ObjectC
methodN()
methodA()
ObjectN
result
ObjectB

Model with DC



A **deployment component** has a number of **concurrent resources**

Deployment Components (1)

A **deployment component** has a number of **concurrent resources**

- These resources are shared between the component's objects

Deployment Components (1)

A **deployment component** has a number of **concurrent resources**

- These resources are shared between the component's objects
- Resources abstract from the number and speed of the physical processors available to the component

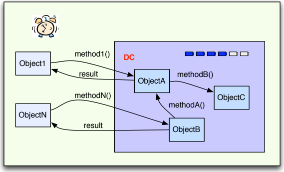
Deployment Components (1)

A **deployment component** has a number of **concurrent resources**

- These resources are shared between the component's objects
- Resources abstract from the number and speed of the physical processors available to the component
- Resources reflect the execution capacity of the deployment component in a time interval

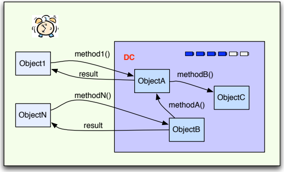
Deployment Components (2)

Consider a deployment component D with r units of processing resources and G objects



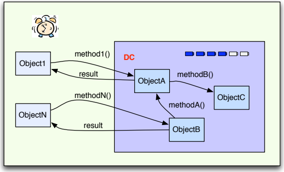
Deployment Components (2)

Consider a deployment component D with r units of processing resources and G objects



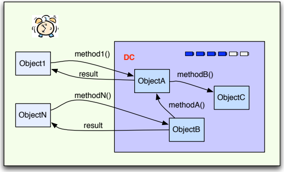
- Any number n of objects in G can execute concurrently ($n \leq r$)

Consider a deployment component D
with r units of processing resources
and G objects



- Any number n of objects in G can execute concurrently ($n \leq r$)
- Let $A \subseteq G$ such that $n = |A|$

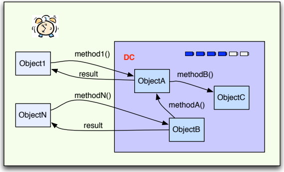
Consider a deployment component D with r units of processing resources and G objects



- Any number n of objects in G can execute concurrently ($n \leq r$)
- Let $A \subseteq G$ such that $n = |A|$
- After one concurrent execution step, D has $r_1 = r - n$ available units of resources.

Deployment Components (2)

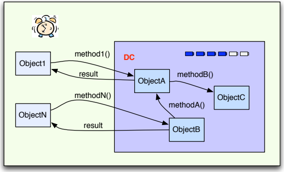
Consider a deployment component D with r units of processing resources and G objects



- Any number n of objects in G can execute concurrently ($n \leq r$)
- Let $A \subseteq G$ such that $n = |A|$
- After one concurrent execution step, D has $r_1 = r - n$ available units of resources.
- If $r_1 > 0$, another execution step can be done (leaving r_2 remaining units of resources available)

Deployment Components (2)

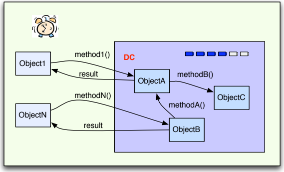
Consider a deployment component D
with r units of processing resources
and G objects



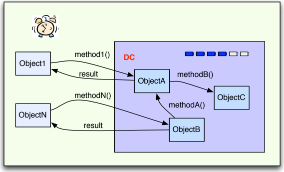
- Any number n of objects in G can execute concurrently ($n \leq r$)
- Let $A \subseteq G$ such that $n = |A|$
- After one concurrent execution step,
  D has $r_1 = r - n$ available units of resources.
- If $r_1 > 0$, another execution step can be done
  (leaving r_2 remaining units of resources available)

Execution inside the time interval stops when no units of resources are available or the objects are blocked

Abs Syntax Extension



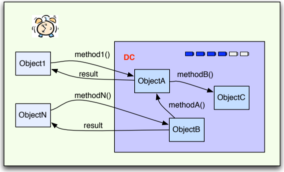
Abs Syntax Extension



Extension of the Syntax of Abs:

- **`now()`** returns the current time

Abs Syntax Extension

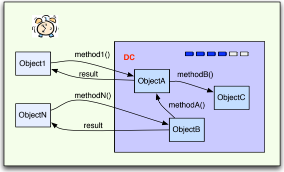


Extension of the Syntax of Abs:

- **now()** returns the current time
- **component**(r) creates a new deployment component

  dc := **component** (r) ;

Abs Syntax Extension



Extension of the Syntax of Abs:

- **now()** returns the current time
- **component**(r) creates a new deployment component

  dc:=**component**(*r*);

- An optional clause in the object creation

  new *C*($\overline{e}$) **in** dc;

Operational Semantics - Extension

The operational semantics of Abs is formalized in rewriting logic and is executable on the Maude tool

Operational Semantics - Extension

The operational semantics of Abs is formalized in rewriting logic and is executable on the Maude tool

A Abs *configuration* (system state) consists of:

Operational Semantics - Extension

The operational semantics of Abs is formalized in rewriting logic and is executable on the Maude tool

A Abs *configuration* (system state) consists of:

Classes, Objects, Futures and Invocation messages

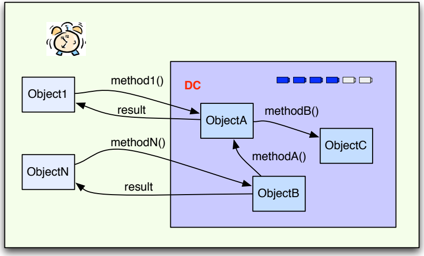
The operational semantics of Abs is formalized in rewriting logic and is executable on the Maude tool

A Abs *configuration* (system state) consists of:

Classes, Objects, Futures and Invocation messages

Extend the configurations with:

- Global clock
  $\langle\ t$:`Clock | Limit:`$l\ \rangle$
- Deployment components
  $\langle\ dc$:`Comp|Free:`r`, Limit:`$l\ \rangle$
- Object attribute **`mycomp`**



Operational Semantics - Extension

Extend the rewriting rules with time, deployment components, and resource consumption

Operational Semantics - Extension

Extend the rewriting rules with time, deployment components, and resource consumption

Some of the statements consume resources when they execute.

Extend the rewriting rules with time, deployment components, and resource consumption

Some of the statements consume resources when they execute.

Simple example:

```
skip;
```

Extend the rewriting rules with time, deployment components, and resource consumption

Some of the statements consume resources when they execute.

Simple example:

```
skip;
```

Old rule:

$$\textbf{rl}\ [\mathit{skip}]:\ \langle o : C \mid \texttt{Pr:}\ \{\bar{l} \mid \texttt{skip;}\ \bar{s}\}\ \rangle$$
$$\longrightarrow\ \langle o : C \mid \texttt{Pr:}\ \{\bar{l} \mid \bar{s}\}\ \rangle\ .$$

Extend the rewriting rules with time, deployment components, and resource consumption

Some of the statements consume resources when they execute.

Simple example:

```
skip;
```

Old rule:

$$\textbf{rl}\ [\mathit{skip}]:\ \langle o : C \mid \texttt{Pr:}\ \{\bar{l} \mid \texttt{skip;}\ \bar{s}\}\ \rangle$$
$$\longrightarrow\ \langle o : C \mid \texttt{Pr:}\ \{\bar{l} \mid \bar{s}\}\ \rangle\ .$$

New rule: skip consumes a resource

$$\textbf{crl}\ [\mathit{skip}]:\ \langle o : C \mid \texttt{Att:}\bar{a}\texttt{, Pr:}\ \{\bar{l} \mid \texttt{skip;}\ \bar{s}\}\ \rangle\ \langle dc\texttt{:Comp}\ \texttt{|Free:}r\rangle$$
$$\longrightarrow\ \langle o : C \mid \texttt{Att:}\bar{a}\texttt{, Pr:}\ \{\bar{l} \mid \bar{s}\}\ \rangle\ \langle dc\texttt{:Comp}\ \texttt{|Free:}r-1\rangle$$
$$\textbf{if}\ dc = \bar{a}[\texttt{mycomp}]\ .$$

Old rule:

$$
\begin{array}{l}
\textbf{crl}\ [\textit{async-call}]:\\
\quad \langle o : C \mid \texttt{Att:}\ \bar{a},\ \texttt{Pr:}\ \{\bar{l} \mid x := e!m(\bar{e});\bar{s}\},\ \texttt{Lcnt:}\ f\ \rangle\\
\longrightarrow \langle o : C \mid \texttt{Att:}\ \bar{a},\ \texttt{Pr:}\ \{\bar{l}\,[x \mapsto n] \mid \bar{s}\},\ \texttt{Lcnt:}\ f+1\ \rangle\\
\quad \texttt{invoc}(\llbracket e \rrbracket_{(\bar{a}\circ\bar{l}),none}, n, m, \llbracket \bar{e} \rrbracket_{(\bar{a}\circ\bar{l}),none})\ \langle n : \texttt{Fut}\ \mid \texttt{Done:false,Value:}\bot\rangle\\
\textbf{if}\ n := \texttt{label}(o, f) \wedge o \neq \llbracket e \rrbracket_{(\bar{a}\circ\bar{l}),none}\ .
\end{array}
$$

New rule consumes resources and evaluates expressions using time:

$$
\begin{array}{l}
\textbf{crl}\ [\textit{async-call}]:\\
\quad \langle o : C \mid \texttt{Att:}\ \bar{a},\ \texttt{Pr:}\ \{\bar{l} \mid x := e!m(\bar{e});\bar{s}\},\ \texttt{Lcnt:}\ f\ \rangle\\
\quad \langle t : \texttt{Clock} \mid\ \rangle\ \langle dc : \texttt{Comp}\ \mid \texttt{Free:}\ r\rangle\\
\longrightarrow \langle o : C \mid \texttt{Att:}\ \bar{a},\ \texttt{Pr:}\ \{\bar{l}\,[x \mapsto n] \mid \bar{s}\},\ \texttt{Lcnt:}\ f+1\ \rangle\\
\quad \langle t : \texttt{Clock} \mid\ \rangle\ \langle dc : \texttt{Comp}\ \mid \texttt{Free:}\ r-1\rangle\\
\quad \texttt{invoc}(\llbracket e \rrbracket^{t}_{(\bar{a}\circ\bar{l}),none}, n, m, \llbracket \bar{e} \rrbracket^{t}_{(\bar{a}\circ\bar{l}),none})\ \langle n : \texttt{Fut}\ \mid \texttt{Done:false,Value:}\bot\rangle\\
\textbf{if}\ n := \texttt{label}(o, f) \wedge o \neq \llbracket e \rrbracket^{t}_{(\bar{a}\circ\bar{l}),none}\ \wedge\ dc = \bar{a}[\texttt{mycomp}]\ .
\end{array}
$$

Operational Semantics - Extension

$$
\begin{array}{l}
\textbf{crl}\ [\textit{progress}]: \\
\quad \{\ cn\ \langle t : \texttt{Clock} \mid \texttt{limit} : \textit{limit} \rangle \} \\
\longrightarrow\ \{\texttt{Adv}(cn)\ \langle t + 1 : \texttt{Clock} \mid \texttt{limit} : \textit{limit} \rangle \} \\
\textbf{if}\ \texttt{canAdv}(cn, t) \wedge t < \textit{limit}\ .
\end{array}
$$

Operational Semantics - Extension

```
crl [progress]:
     {     cn   ⟨t:    Clock | limit: limit⟩ }
⟶   {Adv(cn)  ⟨t + 1: Clock | limit: limit⟩ }
if canAdv(cn, t) ∧ t < limit .
```

Adv(*cn*) resets the free resources of each deployment component to their specified limit.

Operational Semantics - Extension

```
crl [progress]:
      {      cn   ⟨ t:     Clock | limit: limit ⟩ }
⟶    {Adv(cn)  ⟨ t + 1: Clock | limit: limit ⟩ }
if canAdv(cn, t) ∧ t < limit  .
```

Adv(*cn*) resets the free resources of each deployment component to their specified limit.

canAdv(*cn*,*t*) is true if

- no object can do anything and no invocation messages to that object are in the configuration.

Operational Semantics - Extension

```
crl [progress]:
     {     cn   ⟨t:    Clock | limit: limit⟩ }
⟶   {Adv(cn) ⟨t + 1: Clock | limit: limit⟩ }
if canAdv(cn, t) ∧ t < limit .
```

Adv(*cn*) resets the free resources of each deployment component to their specified limit.

canAdv(*cn*,*t*) is true if

- no object can do anything and no invocation messages to that object are in the configuration.

  An object can not do anything if:

  - Its deployment component has run out of resources or
  - All its process are blocked

Otherwise, time cannot advance.

Operational Semantics - Extension

$$
\begin{array}{l}
\textbf{crl}\ [\mathit{progress}]: \\
\quad \{\quad cn\ \langle t:\ \texttt{Clock} \mid \texttt{limit}: \mathit{limit} \rangle \} \\
\longrightarrow\ \{\texttt{Adv}(cn)\ \langle t+1: \texttt{Clock} \mid \texttt{limit}: \mathit{limit} \rangle \} \\
\textbf{if}\ \texttt{canAdv}(cn, t) \wedge t < \mathit{limit}\ .
\end{array}
$$

Adv(cn) resets the free resources of each deployment component to their specified limit.

canAdv(cn,t) is true if

- no object can do anything and no invocation messages to that object are in the configuration.

  An object can not do anything if:

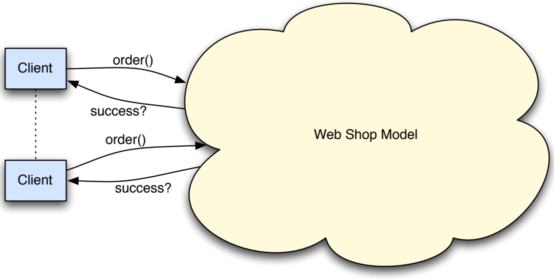
  - Its deployment component has run out of resources or
  - All its process are blocked

Otherwise, time cannot advance.

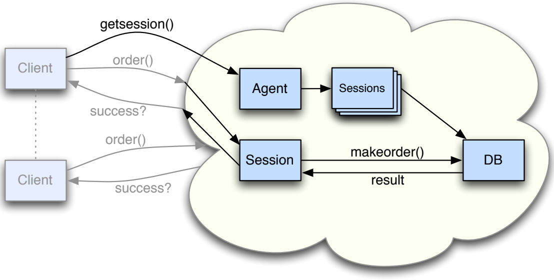
Paper: **Validating Timed Models of Deployment Components with Parametric Concurrency.**

Proc. Int. Conference on Formal Verification of Object-Oriented Software (FoVeOOS) 2010. LNCS 6528, pg. 46–60.

Example: A Shopping Service



Example: A Shopping Service



Example: A Shopping Service - Abs Model

Database

```
interface Database { Bool makeOrder(); }
class Database(Nat min, Nat max) implements Database {
    Bool makeOrder () {
        Time t:=now;
        await now >= t + min;
        return now <= t + max; }
}
```

Example: A Shopping Service - Abs Model

```
interface Database { Bool makeOrder(); }
class Database(Nat min, Nat max) implements Database {
    Bool makeOrder () {
        Time t:=now;
        await now >= t + min;
        return now <= t + max; }
}
```

Session

```
interface Session { Bool order(); }
class Session(Agent agent, Database db) implements Session {
    Bool order() {return db.makeorder(); agent.free(this); }
}
```

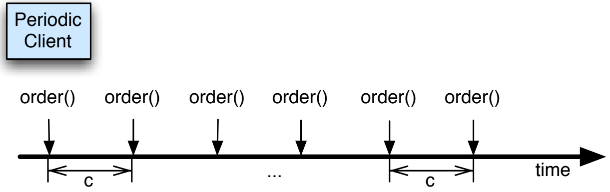
Example: A Shopping Service - Abs Model

```
interface Database { Bool makeOrder(); }
class Database(Nat min, Nat max) implements Database {
    Bool makeOrder () {
        Time t:=now;
        await now >= t + min;
        return now <= t + max; }
}

interface Session { Bool order(); }
class Session(Agent agent, Database db) implements Session {
    Bool order() {return db.makeorder(); agent.free(this); }
}
```

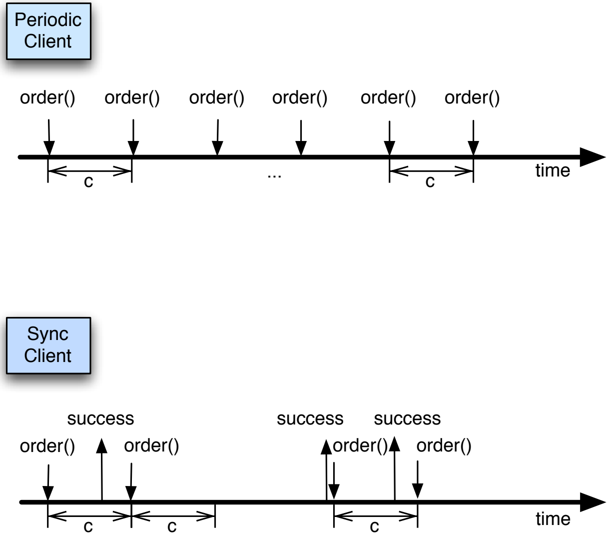
Agent

```
interface Agent { Session getSession(); Void free(Session session); }
class Agent(Database db, Set[Session] sessionPool) implements Agent {
    Session getSession() {
      if isempty(sessionPool) {
            return new Session(this, db); }
      else { session:=choose(sessionPool);
           sessionPool:=remove(session,sessionPool); return session; } }
    Void free(Session session) {sessionPool := add(sessionPool, session);
}
```

Example: Client Behavior



Example: Client Behavior



Example: Client Behavior - Abs Model

Synchronous client

```
class SyncClient(Agent a, Nat c) {
  Void run {
    Time t := now;
    Session s := a.getsession();
    Bool result := s.order();
    await now >= t + c;
    this!run(); } }
```

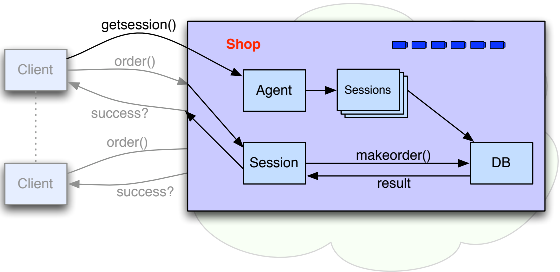
Example: Client Behavior - Abs Model

```
class SyncClient(Agent a, Nat c) {
  Void run {
    Time t := now;
    Session s := a.getsession();
    Bool result := s.order();
    await now >= t + c;
    this!run(); } }
```

Periodic client

```
class PeriodicClient(Agent a, Nat c) {
  Void run {
    Time t := now;
    Session s := a.getsession();
    Fut(Bool) rc := s!order();
    await now >= t + c;
    this!run(); } }
```

Example: Simulation



Example: Simulation and Testing - Abs Model

Different configurations:

```
Void main() {
  Component shop := component(10);
  Database db := new Database(5, 10) in shop;
  Agent a := new Agent(db, {}) in shop;
  SyncClient c := new SyncClient(a, 5); ... }
```

Example: Simulation and Testing - Abs Model

Different configurations:

```
Void main() {
  Component shop := component(10);
  Database db := new Database(5, 10) in shop;
  Agent a := new Agent(db, {}) in shop;
  SyncClient c := new SyncClient(a, 5); ... }
```

or

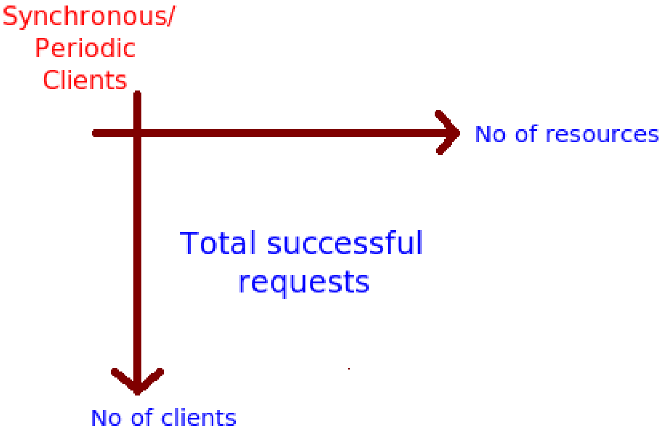
Example: Simulation and Testing - Abs Model

Different configurations:

```
Void main() {
  Component shop := component(10);
  Database db := new Database(5, 10) in shop;
  Agent a := new Agent(db, {}) in shop;
  SyncClient c := new SyncClient(a, 5); ... }
```

or

```
Void main() {
  Component shop := component(10);
  Database db := new Database(5, 10) in shop;
  Agent a := new Agent(db, {}) in shop;
  PeriodicClient c := new PeriodicClient(a, 5); ... }
```

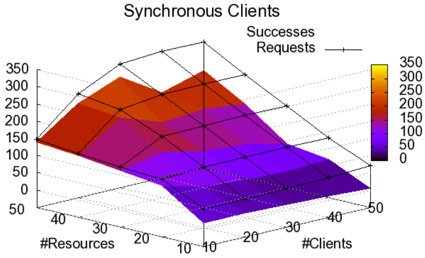


Use Maude as a language interpreter
to simulate the different scenarios

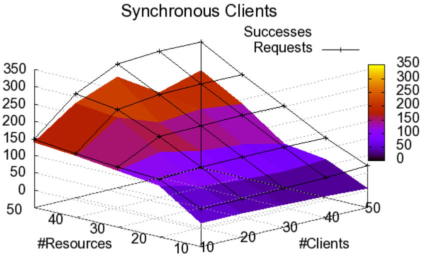
Example: Simulations in the Maude Interpreter - Results

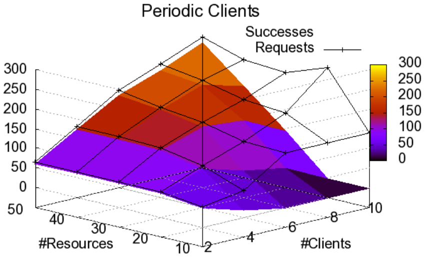
The total and successful requests,
depending on the number of clients and resources

The total and successful requests,
depending on the number of clients and resources

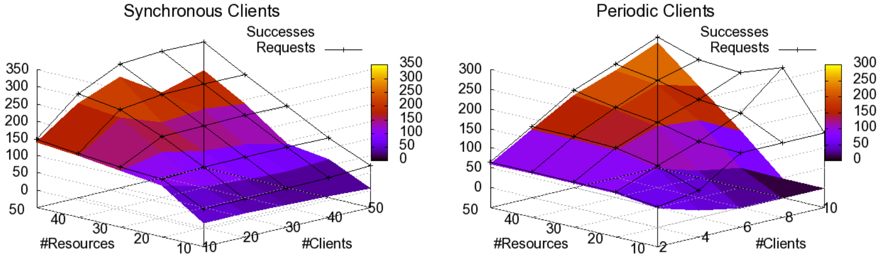


The total and successful requests,
depending on the number of clients and resources



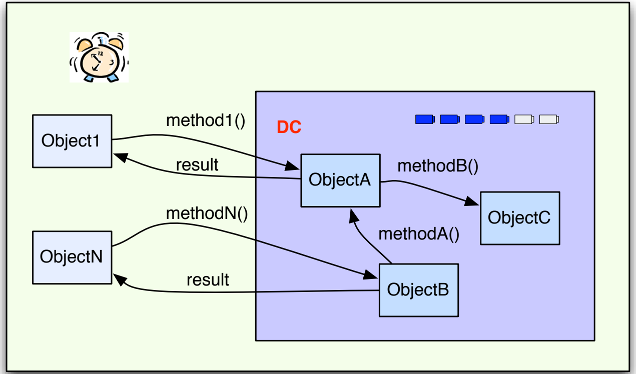


The total and successful requests,
depending on the number of clients and resources

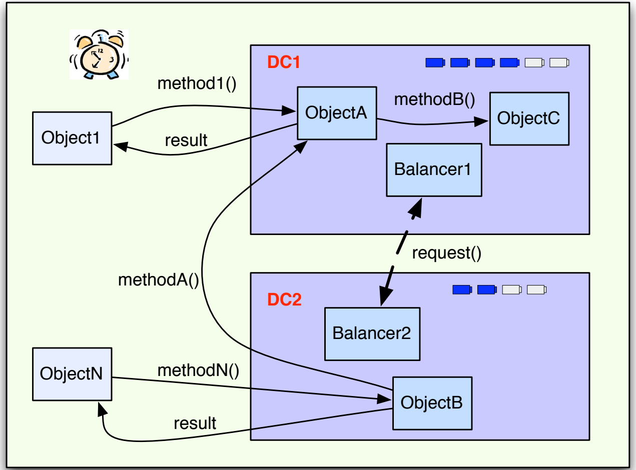


For a larger number of periodic clients, the system becomes unresponsive

Deployment Component



Dynamic Resource Reallocation



Dynamic Resource Reallocation

- Let components and resources be first-class citizens in the language

Dynamic Resource Reallocation

- Let components and resources be first-class citizens in the language
- Now, we can store and pass on components and resource values

Dynamic Resource Reallocation

- Let components and resources be first-class citizens in the language
- Now, we can store and pass on components and resource values

More new expressions and statements in Abs

Dynamic Resource Reallocation

- Let components and resources be first-class citizens in the language
- Now, we can store and pass on components and resource values

More new expressions and statements in Abs

Consider a variable *dc* of type `Component` and *r* of type `Resource`:

Dynamic Resource Reallocation

- Let components and resources be first-class citizens in the language
- Now, we can store and pass on components and resource values

More new expressions and statements in Abs

Consider a variable *dc* of type `Component` and *r* of type `Resource`:

- The expression **`mycomp`** returns *dc* of the object.

Dynamic Resource Reallocation

- Let components and resources be first-class citizens in the language
- Now, we can store and pass on components and resource values

More new expressions and statements in Abs

Consider a variable *dc* of type `Component` and *r* of type `Resource`:

- The expression **`mycomp`** returns *dc* of the object.
- The expression **`available`** returns the number of resources currently allocated to **`mycomp`**

Dynamic Resource Reallocation

- Let components and resources be first-class citizens in the language
- Now, we can store and pass on components and resource values

More new expressions and statements in Abs

Consider a variable *dc* of type `Component` and *r* of type `Resource`:

- The expression **`mycomp`** returns *dc* of the object.
- The expression **`available`** returns the number of resources currently allocated to **`mycomp`**
- The expression **`load`**(e) returns the average number of used resources in **`mycomp`** during the last e time intervals

Dynamic Resource Reallocation

- Let components and resources be first-class citizens in the language
- Now, we can store and pass on components and resource values

More new expressions and statements in Abs

Consider a variable *dc* of type `Component` and *r* of type `Resource`:

- The expression **`mycomp`** returns *dc* of the object.
- The expression **`available`** returns the number of resources currently allocated to **`mycomp`**
- The expression **`load`**(*e*) returns the average number of used resources in **`mycomp`** during the last *e* time intervals
- The statement **`transfer`**(*dc*, *r*) reallocates *r* resources from **`mycomp`** to another component *dc*

Dynamic Resource Reallocation

- Let components and resources be first-class citizens in the language
- Now, we can store and pass on components and resource values

More new expressions and statements in Abs

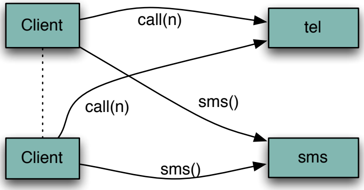
Consider a variable *dc* of type `Component` and *r* of type `Resource`:

- The expression **`mycomp`** returns *dc* of the object.
- The expression **`available`** returns the number of resources currently allocated to **`mycomp`**
- The expression **`load`**(*e*) returns the average number of used resources in **`mycomp`** during the last *e* time intervals
- The statement **`transfer`**(*dc*, *r*) reallocates *r* resources from **`mycomp`** to another component *dc*

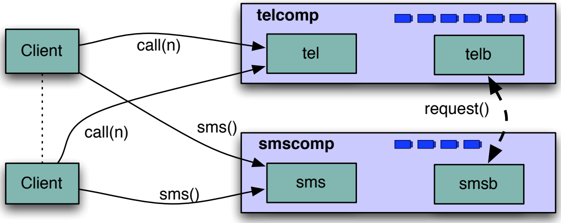
Paper: **Dynamic Resource Reallocation Between Deployment Components.**

Proc. Int. Conference on Formal Engineering Methods (ICFEM) 2010. LNCS 6447, pg. 646–661.

Example: Phone Services



Example: Phone Services



Example: Phone Services - Abs Model

Telephone Service

```
interface TelephoneService { Void call(Int duration); }
class TelephoneService implements TelephoneService {
   Void call(Int duration) {
      Time t; t := now;
      await now >= t + duration; }
}
```

Example: Phone Services - Abs Model

```
interface TelephoneService { Void call(Int duration); }
class TelephoneService implements TelephoneService {
   Void call(Int duration) {
      Time t; t := now;
      await now >= t + duration; }
}
```

SMS Service

```
interface SMSService { Void sendSMS(); }
class SMSService implements SMSService {
   Void sendSMS() { skip; }
}
```

Example: Load Balancing Strategy - Abs Model

The proposed resource-related language-constructors **`available`**, **`load`** and **`transfer`** allow to express different load balancing schemes:

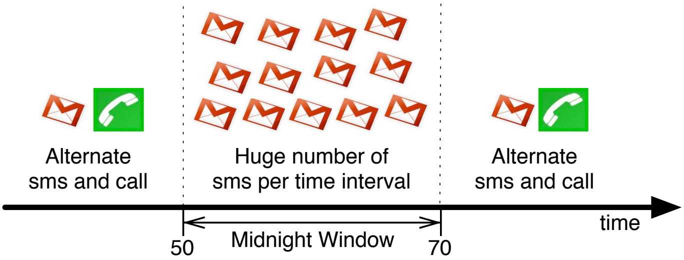
Example: Load Balancing Strategy - Abs Model

The proposed resource-related language-constructors **available**, **load** and **transfer** allow to express different load balancing schemes:

A simple balancer scheme

```
interface Balancer { Void setPartner(Balancer p);
                     Void request(Component comp); }
class Balancer {
  Balancer partner := null;

  Void setPartner(Balancer p) { partner := p; }

  Void request(Component comp) {
    if (load(1) < available−10) {transfer(comp,available/2);} }

  Void run () {
    Time t := now;
    await now > t;
    if (partner≠null ∧ available<load(1)*0.9) {
       partner.request(mycomp);}
    this!run(); }
}
```

Example: The New Year's Eve Client Behavior



Normal behavior of client

```
class NYEbehavior (cycle: Int, ts: TelephoneService, smss: SMSService) {
   Time created := now; Bool call := false;

   Void normalBehavior() {
      Time t := now;
      if (now > created + 50 && now < created + 70) {
         !midnightWindow();
      } else {
         if (call) ts.call(1;) else !smss.sendSMS()
         call := ~ call;
         await now >= t + cycle;
         !normalBehavior(); } }
```

Example: The New Year's Eve Client Behavior

Midnight behavior of client

```
Void midnightWindow() {
   Time t := now;
   Int i := 0;
   if (now > created + 70) {
      !normalBehavior();
   } else {
      while (i < 10) { !smss.sendSMS(); i := i+1; }
      await now > t;
      !midnightWindow(); } }
```

Example: The New Year's Eve Client Behavior

```
Void midnightWindow() {
   Time t := now;
   Int i := 0;
   if (now > created + 70) {
      !normalBehavior();
   } else {
      while (i < 10) { !smss.sendSMS(); i := i+1; }
      await now > t;
      !midnightWindow(); } }
```

Run

```
op run() { !normalBehavior(); } }
```

Example: Simulating and Testing - Abs Model

```
Void main() {
    Component smscomp := component(50);
    Component telcomp := component(50);
```

Example: Simulating and Testing - Abs Model

```
Void main() {
   Component smscomp := component(50);
   Component telcomp := component(50);

   SMSService sms := new SMSService() in smscomp;
   TelephoneService tel := new TelephoneService() in telcomp;
```

Example: Simulating and Testing - Abs Model

```
Void main() {
   Component smscomp := component(50);
   Component telcomp := component(50);

   SMSService sms := new SMSService() in smscomp;
   TelephoneService tel := new TelephoneService() in telcomp;

   Balancer smsb := new Balancer in smscomp;
   Balancer telb := new Balancer in telcomp;
```

Example: Simulating and Testing - Abs Model

```
Void main() {
   Component smscomp := component(50);
   Component telcomp := component(50);

   SMSService sms := new SMSService() in smscomp;
   TelephoneService tel := new TelephoneService() in telcomp;

   Balancer smsb := new Balancer in smscomp;
   Balancer telb := new Balancer in telcomp;

   smsb.setPartner(telb); telb.setPartner(smsb);
```

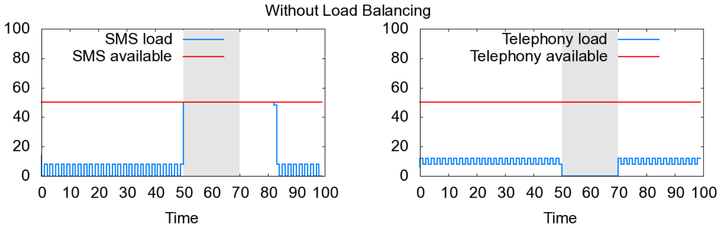
Example: Simulating and Testing - Abs Model

```
Void main() {
   Component smscomp := component(50);
   Component telcomp := component(50);

   SMSService sms := new SMSService() in smscomp;
   TelephoneService tel := new TelephoneService() in telcomp;

   Balancer smsb := new Balancer in smscomp;
   Balancer telb := new Balancer in telcomp;

   smsb.setPartner(telb); telb.setPartner(smsb);

   Client c := new NYEbehavior(1,tel,sms); . . .}
```

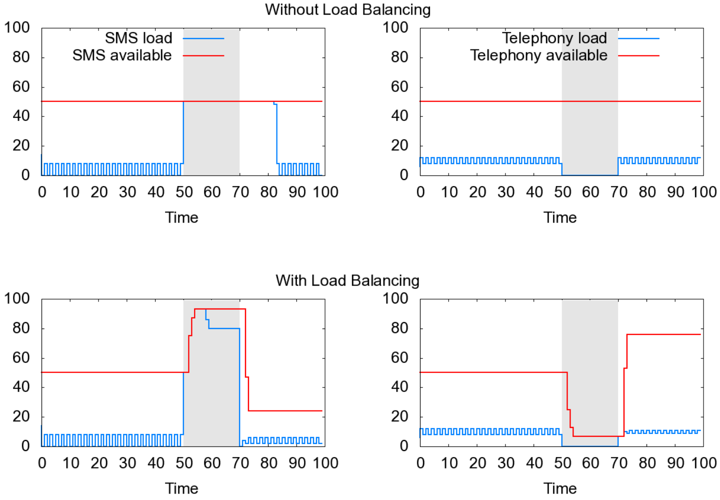
Example: Simulating and Testing - Abs Model

```
Void main() {
   Component smscomp := component(50);
   Component telcomp := component(50);

   SMSService sms := new SMSService() in smscomp;
   TelephoneService tel := new TelephoneService() in telcomp;

  //Balancer smsb := new Balancerinsmscomp;
  //Balancer telb := new Balancerintelcomp;

  //smsb.setPartner(telb); telb.setPartner(smsb);

   Client c := new NYEbehavior(1,tel,sms); . . .}
```

Example: Simulation in the Maude Interpreter

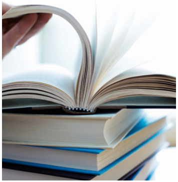


Example: Simulation in the Maude Interpreter



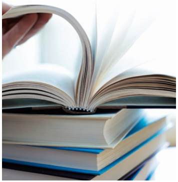
Conclusions

- Modern software is designed to be deployed in different architectures



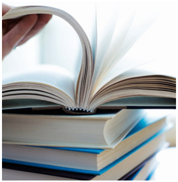
Conclusions

- Modern software is designed to be deployed in different architectures
- Need analysis support which ranges over different deployment scenarios



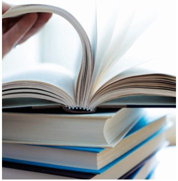
Conclusions

- Modern software is designed to be deployed in different architectures
- Need analysis support which ranges over different deployment scenarios
- We proposed deployment components with parametric concurrent resources



Conclusions

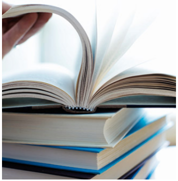
- Modern software is designed to be deployed in different architectures
- Need analysis support which ranges over different deployment scenarios
- We proposed deployment components with parametric concurrent resources



- Abstract notion of resource, reflecting the execution capacity of a component in a given time interval

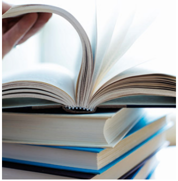
Conclusions

- Dynamic reallocation of resources



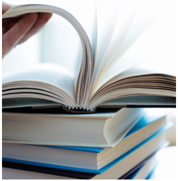
Conclusions

- Dynamic reallocation of resources
- Software controlling allocation and reallocation of resources can be completely separated from the rest of the code



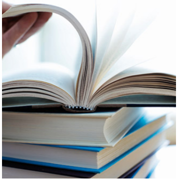
Conclusions

- Dynamic reallocation of resources
- Software controlling allocation and reallocation of resources can be completely separated from the rest of the code



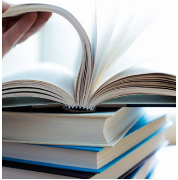
- Different reallocation strategies can be expressed in terms of **`load`**(e), **`available`** and **`transfer`**(dc, r)

Conclusions



- Dynamic reallocation of resources
- Software controlling allocation and reallocation of resources can be completely separated from the rest of the code
- Different reallocation strategies can be expressed in terms of **load**(e), **available** and **transfer**(dc, r)
- It is easy to replace different reallocation strategies for different components

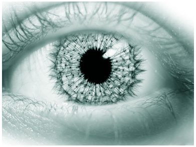
Conclusions



- Dynamic reallocation of resources
- Software controlling allocation and reallocation of resources can be completely separated from the rest of the code
- Different reallocation strategies can be expressed in terms of **`load`**(e), **`available`** and **`transfer`**(dc, r)
- It is easy to replace different reallocation strategies for different components
- Possible to express interesting non-functional system properties

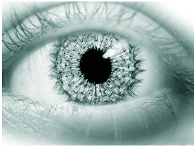
Ongoing/Future Work

- Reallocation between deployment components (eg. load balancing) using object mobility



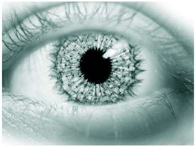
Ongoing/Future Work

- Reallocation between deployment components (eg. load balancing) using object mobility
- Resource adjustments frameworks using hierarchical strategies



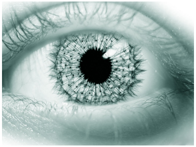
Ongoing/Future Work

- Reallocation between deployment components (eg. load balancing) using object mobility
- Resource adjustments frameworks using hierarchical strategies
- Stronger analysis methods
  - Symbolic analysis
  - Static analysis



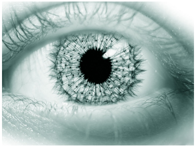
Ongoing/Future Work

- Reallocation between deployment components (eg. load balancing) using object mobility
- Resource adjustments frameworks using hierarchical strategies
- Stronger analysis methods
  - Symbolic analysis
  - Static analysis
- Memory resources for deployment components



Ongoing/Future Work

- Reallocation between deployment components (eg. load balancing) using object mobility
- Resource adjustments frameworks using hierarchical strategies
- Stronger analysis methods
  - Symbolic analysis
  - Static analysis
- Memory resources for deployment components
- Scheduling
  - Priority scheduling: Processes can dynamically increase or decrease in priority according to their waiting time
  - Deadlines to method calls



THANK YOU