Verified and Efficient Reconfigurations of Component-Based Systems

Simon Robillard

October 22, 2020
Verified and Efficient Reconfigurations of Component-Based Systems

Introduction

About me

PhD thesis

- defended in May 2019
- Chalmers University of Technology (Sweden)
- supervised by Laura Kovács
- work on the first-order theorem prover Vampire

First-order theorem provers

1. as a developer: add theory reasoning (inductive data types)
2. as a user: program verification (loop analysis)
STACK team

Team goals

- develop a tightly-coupled software stack for utility computing infrastructures (cloud, fog, edge)
- adapted abstractions for every level

Post-doc

- since July 2019
- with Hélène Coullon
- goal: bring some formal methods to the team
The promises of cloud computing

Global scale
The benefits of cloud computing services include the ability to scale elastically. In cloud speak, that means delivering the right amount of IT resources—for example, more or less computing power, storage, bandwidth—right when they’re needed, and from the right geographic location.

Flexibility
Users can scale services to fit their needs, customize applications and access cloud services from anywhere with an internet connection.

Agility
The cloud gives you easy access to a broad range of technologies so that you can innovate faster and build nearly anything that you can imagine. You can quickly scale up resources as you need them. All infrastructure services, such as compute, storage, and databases, to distributed big data, machine learning, data insights, and much more.

You can design technology services in a matter of minutes, and get from idea to demonstration server in a matter of hours, much faster than before. This gives you the freedom to experiment, test new ideas to differentiate customer experiences, and position your business.

Elasticity
With cloud computing, you don’t have to overprovision resources up front. You can scale with your level of business activity by the hour. Indeed, you can provision or de-provision resources instantly to grow or shrink capacity as your business needs change.
Verified and Efficient Reconfigurations of Component-Based Systems

Introduction

The promises of cloud computing

we study **component-based systems**

- broad notion of “software components”
  - independent, replacable, reusable units
  - well-defined interface
- working together in an assembly
The promises of cloud computing

we study **component-based systems**

► **broad notion of “software components”**
  - independent, replacable, reusable units
  - well-defined interface

► **working together in an assembly**

and in particular their **reconfigurations**

► **topology of assembly + component states**
  - deployment
  - scaling
  - migration
  - maintenance
  - and more...
Verified and Efficient Reconfigurations of Component-Based Systems

Introduction

The promises of cloud computing

Deploy globally in minutes
With the cloud, you can expand to new geographic regions and deploy globally in minutes. For example, AWS has infrastructure all over the world, so you can deploy your application in multiple physical locations with just a few clicks. Pacing applications in closer proximity to end users reduces latency and improves their experience.

Efficiency
Enterprise users can get applications to market quickly, without worrying about underlying infrastructure costs or maintenance.

Performance
The biggest cloud computing services run on a worldwide network of secure data centers, which are regularly upgraded to the latest generation of fast and efficient computing hardware. This offers several benefits over a single corporate data center, including reduced network latency for applications and greater economies of scale.

Speed
Most cloud computing services are provided on a self-service and on-demand basis, so even vast amounts of computing resources can be provisioned in minutes, typically with just a few mouse clicks, giving businesses a lot of flexibility and taking the pressure off capacity planning.
The promises of cloud computing

- **structured parallelism**
  - declarative approach
    - easier to design
    - easier to analyze and verify
  - expressive and **fine-grained** model
  - required for performance
    - inter-component parallelism
    - intra-component parallelism
Verified and Efficient Reconfigurations of Component-Based Systems

Introduction

The promises of cloud computing

Reliability
Cloud computing makes data backup, disaster recovery, and business continuity easier and less expensive because data can be mirrored at multiple redundant sites on the cloud provider’s network.
The promises of cloud computing

- improve development practices
  - model with formally-defined semantics
  - analysis tools to assist design
    - performance prediction
    - critical path analysis
    - parallelism analysis
    - **automated verification**

Reliability
Cloud computing makes data backup, disaster recovery, and business continuity easier and less expensive because data can be mirrored at multiple redundant sites on the cloud provider's network.
Section 1

Concerto: a Model for Efficient Reconfigurations
Concerto: goals and philosophy

A model for reconfigurations in component-based systems

- represent the lifecycle of components
  - non-functional aspect
  - in this talk, component = control component
- coordinate reconfiguration actions
  - e.g. starting/stopping VM, downloading images, installing/updating software...
Concerto: goals and philosophy

A model for reconfigurations in component-based systems

▶ represent the lifecycle of components
  • non-functional aspect
  • in this talk, component = control component
▶ coordinate reconfiguration actions
  • e.g. starting/stopping VM, downloading images, installing/updating software...

Reliability

▶ formally-defined semantics
▶ tools to assist during design
▶ formal methods

Performance

▶ structured parallelism
▶ declarative approach
▶ Python implementation
Verified and Efficient Reconfigurations of Component-Based Systems
Concerto: a Model for Efficient Reconfigurations

Performance through parallelism

- level 1: multiple nodes, same action
- no dependencies declared
- procedural execution order
- Ansible
Performance through parallelism

- level 2: non-dependent components
- dependencies at the component level
- Deployware, Tosca
Performance through parallelism

- level 3: inter-component
- dependencies at the task level
- Aeolus
Verified and Efficient Reconfigurations of Component-Based Systems
Concerto: a Model for Efficient Reconfigurations

Performance through parallelism

- level 4: intra-component
- internal task dependencies
- Concerto
Performance through parallelism

- level 4: intra-component
- internal task dependencies
- Concerto

Parallel execution requires precise description of dependencies
Anatomy of a Concerto component

**Internal net**

- places = milestones
- transitions = actions to perform
  - concretely: scripts are attached to transitions
  - in the model: the exact nature/effects of actions is not represented, only coordination

**Ports**

- data or service exchanges
  - use ports = requirements
  - provide ports = provisions
- can be turned on/off (e.g. service not required anymore, data becoming obsolete)
Execution

“Petri net” style of semantics

- can be in multiple places at once
- transitions not atomic
- can execute multiple transitions at once

Coordination via ports

- use port need to be provided before reaching places
- places cannot be left while provide ports are used
Execution

“Petri net” style of semantics

- can be in multiple places at once
- transitions not atomic
- can execute multiple transitions at once

Coordination via ports

- use port need to be provided before reaching places
- places cannot be left while provide ports are used
Execution

“Petri net” style of semantics
- can be in multiple places at once
- transitions not atomic
- can execute multiple transitions at once

Coordination via ports
- use port need to be provided before reaching places
- places cannot be left while provide ports are used
Execution

“Petri net” style of semantics
- can be in multiple places at once
- transitions not atomic
- can execute multiple transitions at once

Coordination via ports
- use port need to be provided before reaching places
- places cannot be left while provide ports are used
Execution

“Petri net” style of semantics

- can be in multiple places at once
- transitions not atomic
- can execute multiple transitions at once

Coordination via ports

- use port need to be provided before reaching places
- places cannot be left while provide ports are used
Execution

“Petri net” style of semantics
- can be in multiple places at once
- transitions not atomic
- can execute multiple transitions at once

Coordination via ports
- use port need to be provided before reaching places
- places cannot be left while provide ports are used
Execution

“Petri net” style of semantics

▷ can be in multiple places at once
▷ transitions not atomic
▷ can execute multiple transitions at once

Coordination via ports

▷ use port need to be provided before reaching places
▷ places cannot be left while provide ports are used
Execution

“Petri net” style of semantics
- can be in multiple places at once
- transitions not atomic
- can execute multiple transitions at once

Coordination via ports
- use port need to be provided before reaching places
- places cannot be left while provide ports are used
Execution

“Petri net” style of semantics
- can be in multiple places at once
- transitions not atomic
- can execute multiple transitions at once

Coordination via ports
- use port need to be provided before reaching places
- places cannot be left while provide ports are used
Adapting to new conditions through behaviors

- allow components to act in various ways by changing their internal net
- behaviors $\simeq$ API for components
  - requests to execute a behavior are asynchronous
  - when there is no more transition to execute in a behavior: switch to the next requested
Concerto in practice: for the component developer

Declaring a control component

- new Python (sub)class
- Python function attached to transition
- easy to wrap legacy software in control component

```python
class MyComponent(Component):
    def create(self):
        self.places = ['uninstalled', 'installed', 'running']

        self.transitions = {
            'install': ('uninstalled', 'installed', 'bvh_start', self.install),
            'start': ('installed', 'running', 'bvh_start', self.start),
            'stop': ('running', 'installed', 'bvh_stop', self.stop)
        }

        self.dependencies = {'provide_port': (DepType.PROVIDE, ['running'])}

        self.initial_place = 'uninstalled'

    def install(self):
        remote = SSHClient()
        remote.connect(host, user, pwd)
        remote.exec_command(cmd)
        ...

    def start(self):
        ...

    def stop(self):
        ...
```
Concerto in practice: for the system administrator

```python
class MyAssembly(Assembly):
    def deploy(self):
        self.add_component('client', Client())
        self.add_component('server', Server())
        self.connect('client', 'use_server_ip', 'server', 'provide_ip')
        self.connect('client', 'use_service', 'server', 'provide_service')
        self.push_b('client', 'install_start')
        self.push_b('server', 'deploy')
        self.wait_all()
        self.synchronize()

    def maintain(self):
        self.push_b('client', 'stop')
        self.push_b('server', 'stop')
        self.push_b('client', 'install_start')
        self.push_b('server', 'deploy')
        self.wait_all()
        self.synchronize()
```

Reconfiguration programs

- **commands**
  - create/delete component instance
  - connect/disconnect component instances
  - request behavior
  - synchronization (wait until a component instance has finished all behavior requests)

- **deep-embedding + interpreter**
Additional tools: performance prediction

- **input**
  - reconfiguration program
  - time estimations for transitions

- **generate weighted dependency graph**

- **output**
  - critical path in dependency graph
  - reconfiguration time (assuming hardware can execute as many concurrent threads as needed)
Verified and Efficient Reconfigurations of Component-Based Systems
Concerto: a Model for Efficient Reconfigurations

Additional tools: model-checking

- targets Madeus (subset/predecessor of Concerto, limited to deployment)
- translation to (time) Petri nets
- check qualitative properties
  - invariant
  - final state
  - sequentiality
- check quantitative properties
  - execution time (interval)
  - maximum number of concurrent threads
Section 2

Verified Reconfigurations
Verified and Efficient Reconfigurations of Component-Based Systems

Verified Reconfigurations

Reconfiguration program                  Property

Event ordering problem

SMT solver     FO prover     Finite model finder     SAT solver     CSP solver
Event ordering problems

Problem
- events $E$, including $init$ and $final$.
- boolean attributes over events $A$
- first-order constraints with predicate symbols $occurs$, $succ$ and $before$

Solution
- subset of events that do occur (including $init$ and $final$)
- total ordering of occurring events
- valuation of attributes for occurring events
- must respect constraints
Event ordering problem example: opening a door

- events *init*, *final*, *unlock* and *open*
- attributes *locked* and *opened*
- constraints for post-conditions

\[ \neg \text{locked}(\text{unlock}) \]

- constraints for pre-conditions

\[ \forall e \left( \text{succ}(e, \text{open}) \implies \neg \text{locked}(e) \land \neg \text{opened}(e) \right) \]

- constraints indicating that attributes are not affected by some events

\[ \forall e_1 e_2 \left( \text{succ}(e_1, e_2) \land e_2 \neq \text{unlock} \implies \left( \text{locked}(e_1) \iff \text{locked}(e_2) \right) \right) \]
Solutions

Event Ordering Problem

SMTLIB translation

- constraints
- axioms for `succ`, `before` and `occurs`
- (non recursive) datatype to represent finite domain of events
Enumerating solutions using a model finder or SMT solver

- \textit{init} \prec \textit{final} with same valuation of attributes for both events
- \textit{init} \prec \textit{unlock} \prec \textit{final} with \texttt{locked} = \{\textit{init}\} and \texttt{opened} = \emptyset
- \ldots
Verified and Efficient Reconfigurations of Component-Based Systems

Verified Reconfigurations

Solutions

Verifying properties by refutation

- e.g. invariant

\[ \forall e \ (\text{occurs}(e) \implies \neg(\text{opened}(e) \land \text{locked})) \]

- add the negation of the constraint
  - use SMT solver or FO prover to refute: the property is verified
  - if the resulting problem is satisfiable, a SMT solver or model finder returns a counter-example sequence of events
Expressing temporal properties with quantifiers

- final state
  \[ P(\text{final}) \]

- invariant
  \[ \forall x. \text{occurs}(x) \implies Q(x) \]

- intermediate state
  \[ \exists x. \text{occurs}(c) \land R(x) \]

- sequence of intermediate states
  \[ \exists x_1 x_2 \ldots x_{n-1} x_n. \text{before}(x_1, x_2) \land \cdots \land \text{before}(x_{n-1}, x_n) \land T_1(x_1) \land \cdots \land T_n(x_n) \]
Modeling a component execution

Choice of timeline attributes

- coarse abstraction \(\Rightarrow\) fewer events to consider
- abstraction guided by component-based nature of model
  - components as black boxes
  - attributes corresponding to ports
  - one attribute indicating whether the behavior queue is empty (needed for synchronization barrier)
Verified and Efficient Reconfigurations of Component-Based Systems

Verified Reconfigurations

Modeling a component execution

Selection of relevant events

- reaching (or leaving) a place: only if it can (de)activate a port
- reaching the final state of the behavior
Verified and Efficient Reconfigurations of Component-Based Systems
Verified Reconfigurations

Modeling a component execution

Constraints

- order in which places can be reached (given by the graph of the internal net)
- pre-conditions related to ports
  - entering place connected to use port
  - leaving place connected to provide port
Modeling a component execution

- **attributes**
  - `config_info_active`
  - `service_active`
  - `no_pending_request`

- **events**
  - `reach_installed`
  - `reach_running`

- **constraints**
  - `reach_running` occurs after `reach_installed`
  - `reach_installed` can occur only if the use port `config_info` is connected to an active provide port
  - Constraints describing which events affect which attributes
Modeling a reconfiguration program

Attributes
- connections
- component creation do not need to be represented through attributes (well-formedness can be checked statically)

Events and constraints
- one event per instruction
- if the instruction is a behavior request, add events mentioned in previous slide
- constraints on order of instructions
- pre-condition for synchronization barrier, disconnection, deletion of components

```python
self.add_component('client', Client())
self.add_component('server', Server())
self.connect('client', 'use_server_ip', 'server', 'provide_ip')
self.connect('client', 'use_service', 'server', 'provide_service')
self.push_b('client', 'install_start')
self.push_b('server', 'deploy')
self.wait_all()
```
Substitutability of components

Can we replace component $C_1$ by component $C_2$?

- given: correspondance from ports of $C_1$ to ports of $C_2$
- starting configuration (internal state) of both components

Check that any execution trace of $C_1$ is an execution of $C_2$
Substitutability of components

Can we replace component $C_1$ by component $C_2$?

- given: correspondance from ports of $C_1$ to ports of $C_2$
- starting configuration (internal state) of both components

Check that any execution trace of $C_1$ is an execution of $C_2$

Comparing timelines with different events

- only consider attributes
- consider intervals, not events
Timeline equivalence

Equivalence up to homomorphism = exists relation $H$ such that

- $\forall xx'. (x, x') \in H \implies \text{attributes}(x) = \text{attributes}(x')$
- $\forall xyx'y'. (x, x') \in H \land (y, y') \in H \implies (x \leq y \iff x' \leq y')$
- every event occurring in a timeline is $H$-related to an event occurring in the other

$$T = a \ b \ c \ d \ e \ f \ g \ ...$$

$$T' = a' \ b' \ c' \ d' \ e' \ f' \ g' \ ...$$
Timeline equivalence

Equivalence up to homomorphism = exists relation $H$ such that

- $\forall xx'. (x, x') \in H \implies \text{attributes}(x) = \text{attributes}(x')$
- $\forall xyx'y'. (x, x') \in H \land (y, y') \in H \implies (x \leq y \iff x' \leq y')$
- every event occurring in a timeline is $H$-related to an event occurring in the other

\[
T = a \quad b \quad c \quad d \quad e \quad f \quad g \ldots
\]
\[
T' = a' \quad b' \quad c' \quad d' \quad e' \quad f' \quad g' \ldots
\]
Timeline equivalence

Equivalence up to homomorphism = exists relation $H$ such that

- $\forall xx'. (x, x') \in H \implies \text{attributes}(x) = \text{attributes}(x')$
- $\forall xyx'y'. (x, x') \in H \land (y, y') \in H \implies (x \leq y \iff x' \leq y')$
- every event occurring in a timeline is $H$-related to an event occurring in the other

Proving soundness of system abstraction w.r.t. original Concerto semantics

- also up to homomorphism
- + abstraction of model through attributes
Proving component substitutability

\[ \forall T_1 T_2. \text{axioms}(T_1) \land \text{semantics}_{C_1}(T_1) \land \text{homomorphic}(T_1, T_2) \implies \text{semantics}_{C_2}(T_2) \]
Proving component substitutability

∀ \ T_1 \ T_2. \ axioms( T_1 ) \land \ semantics_{C_1}( T_1 ) \land homomorphic( T_1, T_2 ) \implies \ semantics_{C_2}( T_2 )

Relaxed axiomatization

- unlike verification, where all possible events are known...
- ...here we can't fix the size of the domain
In summary

Concerto
- a model for reconfiguration in component-based systems
- structured parallelism
- formally-defined semantics

Event-based verification
- verification of executions
- substitutability (relational hyperproperties)
- applications beyond component-based models
The next step: synthesis

- initial state +
  - target state(s) +
  - description of available components

reconfiguration generator

- candidate program
- additional constraints

reconfiguration verifier

- counter-example execution
- correct-by-construction script