Run-time Failure Forecasting for Dynamically Evolving Systems

on-going work

Andrea Polini

joint work with: **Maria Rita Di Berardini, Henry Muccini and Pengcheng Zhang**

ISTI / UNICAM

CINA – Kick-off meeting

Pisa – February 05, 2013

Outline

Motivations

How are software systems composed nowadays?

- more and more dynamically at run-time
- more and more in fast and short living compositions
- more and more in a multi-players settings

Motivations

How are software systems composed nowadays?

- more and more dynamically at run-time
- more and more in fast and short living compositions
- more and more in a multi-players settings

Consequences

- it is not possible to know all the possible configurations of a software system
- Most of the effort spent in off-line verification activites would be “wasted”
- volatile configurations do not leave much time for effort prone verification activities

Motivations

How are software systems composed nowadays?

- more and more dynamically at run-time
- more and more in fast and short living compositions
- more and more in a multi-players settings

Consequences

- it is not possible to know all the possible configurations of a software system
- Most of the effort spent in off-line verification activites would be "wasted"
- volatile configurations do not leave much time for effort prone verification activities

Our Hypothesis

In such settings "traditional" off-line verification techniques should be complemented with novel "on-line" approaches.

Assumptions

We wanted to derive an approach for dynamically evolving systems able to predict (potential) future failure. Main assumptions:

- Availability of design time models of components prescribed usage
- Capability to monitor interactions among components
- Capability to relate monitored "fact" and information in the models

Assumptions

We wanted to derive an approach for dynamically evolving systems able to predict (potential) future failure. Main assumptions:

- Availability of design time models of components prescribed usage
- Capability to monitor interactions among components
- Capability to relate monitored "fact" and information in the models

Cassandra

This is the approach and the tool we defined. It composes (on-the-fly) available component models and it checks if in the near future something bad can happen

Interface Automata (L. de Alfaro, T. Henzinger)

Interface Automata

An interface automaton is a tuple $P = \langle V_P, V_P^{init}, \mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H, \mathcal{T}_P \rangle$ where:

- V_P is a set of states and V_P^{init} is a set of initial states that contains at most one state.
- $\mathcal{A}_P^I, \mathcal{A}_P^O$ and $\mathcal{A}_P^H$ are mutually disjoint sets of input, output and internal actions. We define $\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O \cup \mathcal{A}_P^H$. P is closed if it has only internal actions (i.e. if $\mathcal{A}_P^I = \mathcal{A}_P^O = \emptyset$); otherwise P is open.
- $\mathcal{T}_P \subseteq V_P \times \mathcal{A}_P \times V_P$ is a set of steps.

Composition

Two interface automata P and Q are composable if $\mathcal{A}_P^H \cap \mathcal{A}_Q = \mathcal{A}_Q^H \cap \mathcal{A}_P = \emptyset$ and $\mathcal{A}_P^I \cap \mathcal{A}_Q^I = \mathcal{A}_P^O \cap \mathcal{A}_Q^O = \emptyset$. Note that, if P and Q are composable, then $\text{shared}(P, A) = \mathcal{A}_P \cap \mathcal{A}_Q = (\mathcal{A}_P^I \cap \mathcal{A}_Q^O) \cup (\mathcal{A}_Q^I \cap \mathcal{A}_I^O)$.
Interface automata interact through the synchronization of shared input and output actions, and asynchrounously interleave all the other actions.

Composition

Two interface automata P and Q are composable if $\mathcal{A}_P^H \cap \mathcal{A}_Q = \mathcal{A}_Q^H \cap \mathcal{A}_P = \emptyset$ and $\mathcal{A}_P^I \cap \mathcal{A}_Q^I = \mathcal{A}_P^O \cap \mathcal{A}_Q^O = \emptyset$. Note that, if P and Q are composable, then $\text{shared}(P, A) = \mathcal{A}_P \cap \mathcal{A}_Q = (\mathcal{A}_P^I \cap \mathcal{A}_Q^O) \cup (\mathcal{A}_Q^I \cap \mathcal{A}_I^O)$.
Interface automata interact through the synchronization of shared input and output actions, and asynchrounously interleave all the other actions.

Illegal state

An illegal state corresponds to a state in which a shared action is available as output in one of the two composed automata and not as an input in the other (illegal action).

Composition

Two interface automata P and Q are composable if $\mathcal{A}_P^H \cap \mathcal{A}_Q = \mathcal{A}_Q^H \cap \mathcal{A}_P = \emptyset$ and $\mathcal{A}_P^I \cap \mathcal{A}_Q^I = \mathcal{A}_P^O \cap \mathcal{A}_Q^O = \emptyset$. Note that, if P and Q are composable, then $\text{shared}(P, A) = \mathcal{A}_P \cap \mathcal{A}_Q = (\mathcal{A}_P^I \cap \mathcal{A}_Q^O) \cup (\mathcal{A}_Q^I \cap \mathcal{A}_I^O)$.
Interface automata interact through the synchronization of shared input and output actions, and asynchrounously interleave all the other actions.

Illegal state

An illegal state corresponds to a state in which a shared action is available as output in one of the two composed automata and not as an input in the other (illegal action).

Legal Environment

Given two composable interace automata P and Q, a legal environment is an environment such that no illegal state can be reached in the composition with the environment.

Composition

Two interface automata P and Q are composable if $\mathcal{A}_P^H \cap \mathcal{A}_Q = \mathcal{A}_Q^H \cap \mathcal{A}_P = \emptyset$ and $\mathcal{A}_P^I \cap \mathcal{A}_Q^I = \mathcal{A}_P^O \cap \mathcal{A}_Q^O = \emptyset$. Note that, if P and Q are composable, then $\text{shared}(P, A) = \mathcal{A}_P \cap \mathcal{A}_Q = (\mathcal{A}_P^I \cap \mathcal{A}_Q^O) \cup (\mathcal{A}_Q^I \cap \mathcal{A}_I^O)$.
Interface automata interact through the synchronization of shared input and output actions, and asynchrounously interleave all the other actions.

Illegal state

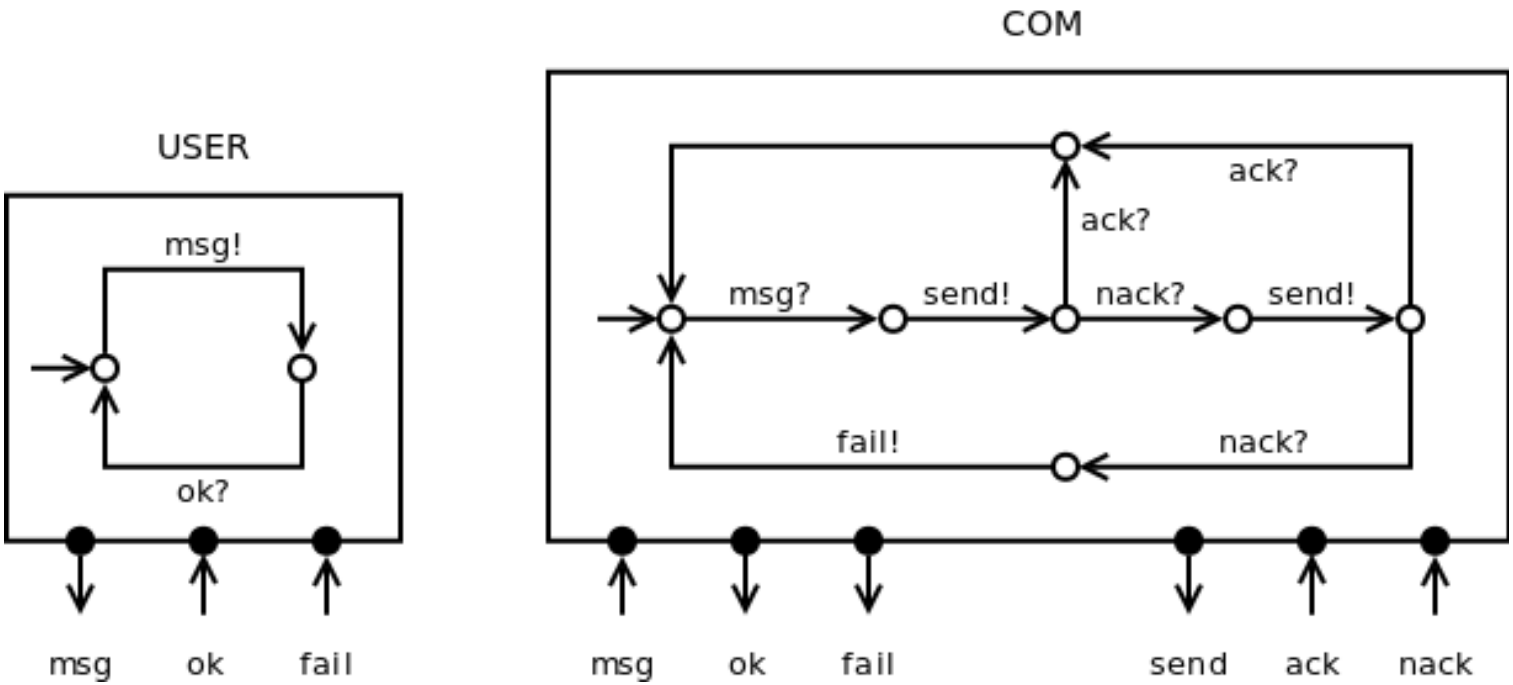
An illegal state corresponds to a state in which a shared action is available as output in one of the two composed automata and not as an input in the other (illegal action).

Legal Environment

Given two composable interace automata P and Q, a legal environment is an environment such that no illegal state can be reached in the composition with the environment.

Two automata are compatible if it exists a legal environment

Interface Automata Example



On the same track but slightly different

In our approach we consider a slightly revised version of Interface Automata:

- Illegal states are those states reached executing an illegal action.
  - They intend to represent wrong invocations made on available services
- Warning states are those reached by non shared output actions
  - They intend to represent invocations to services not anymore available, e.g. due to the removal of a component

Different interpretation due to different objectives

Monitoring and failure detection vs. Verification

Cassandra

Cassandra takes in input the following information

1. an integer l greater than 1 (look-ahead)
2. interface automata for the integrated components

As a result Cassandra engages in the following steps:

1. It derives a composition of the inteface automata looking l-steps ahead
2. In case a warning or illegal state is detected (i.e. it can be reached from the current state within l steps) the trace leading to the "guilty" state is returned
3. it continuously observe the real interactions happening among the running components and keep the model up-to-date; still looking l-steps ahead.
4. in case of insertion/removal of a component the model is updated

Cassandra

Cassandra takes in input the following information

1. an integer l greater than 1 (look-ahead)
2. interface automata for the integrated components

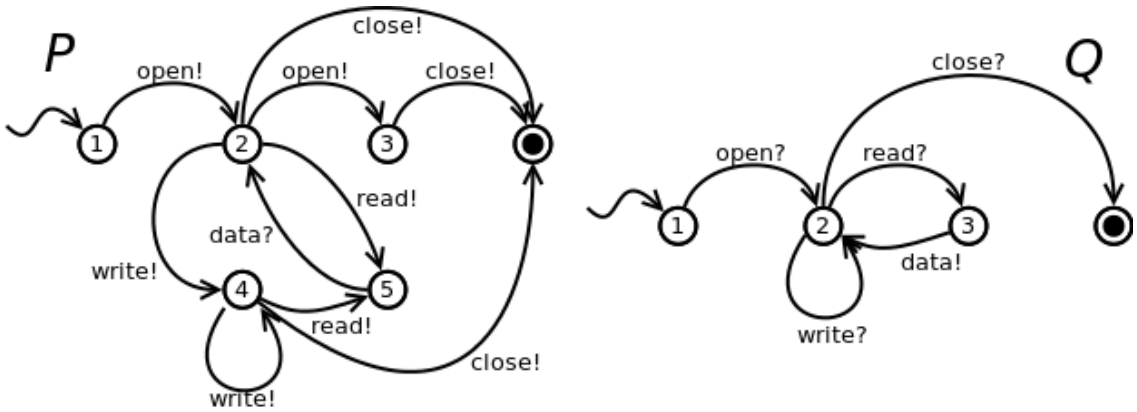
As a result Cassandra engages in the following steps:

1. It derives a composition of the inteface automata looking l-steps ahead
2. In case a warning or illegal state is detected (i.e. it can be reached from the current state within l steps) the trace leading to the "guilty" state is returned
3. it continuously observe the real interactions happening among the running components and keep the model up-to-date; still looking l-steps ahead.
4. in case of insertion/removal of a component the model is updated

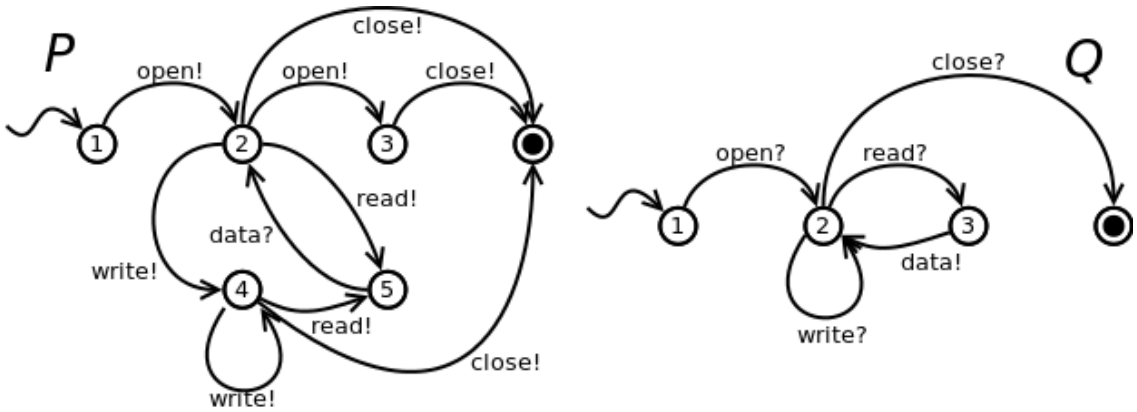
"Cassandra's motto"

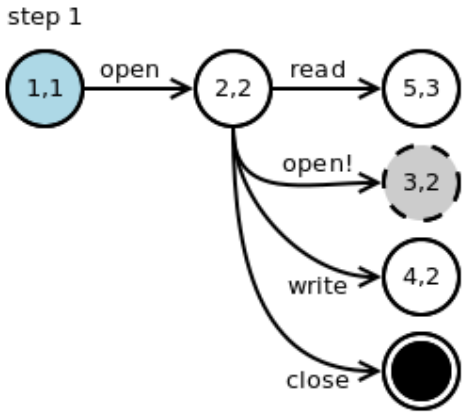
Integration should never be impeded a priori, something good can always happen

Sketching the idea ($l = 2$)

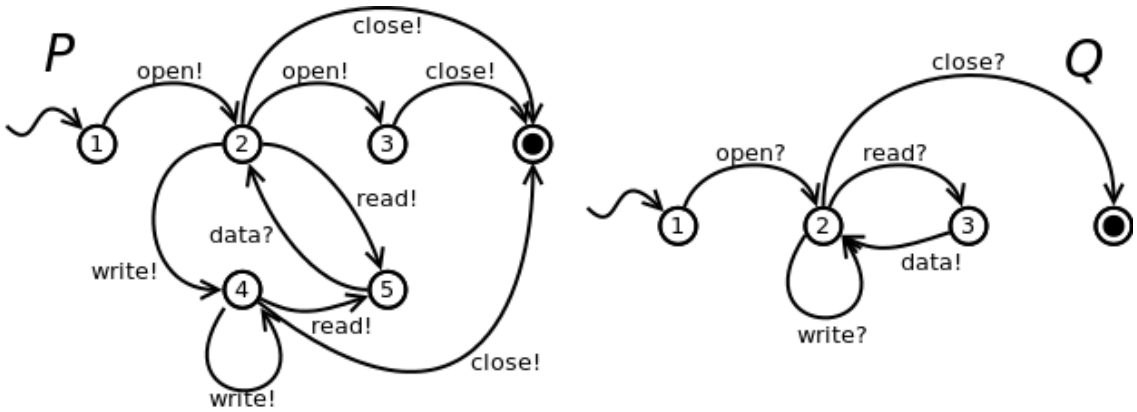


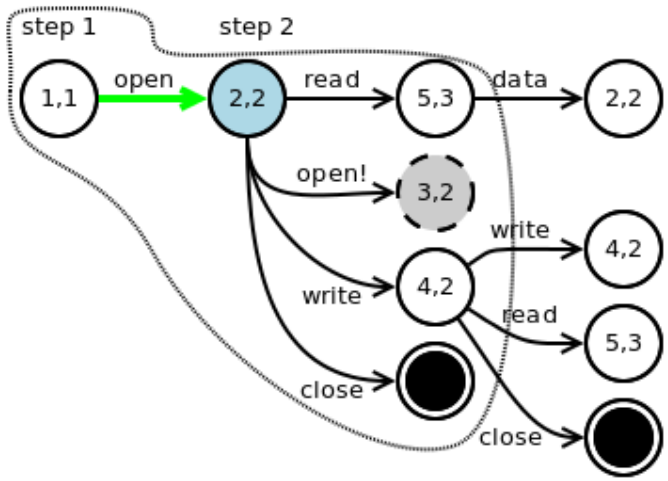
Sketching the idea ($l = 2$)



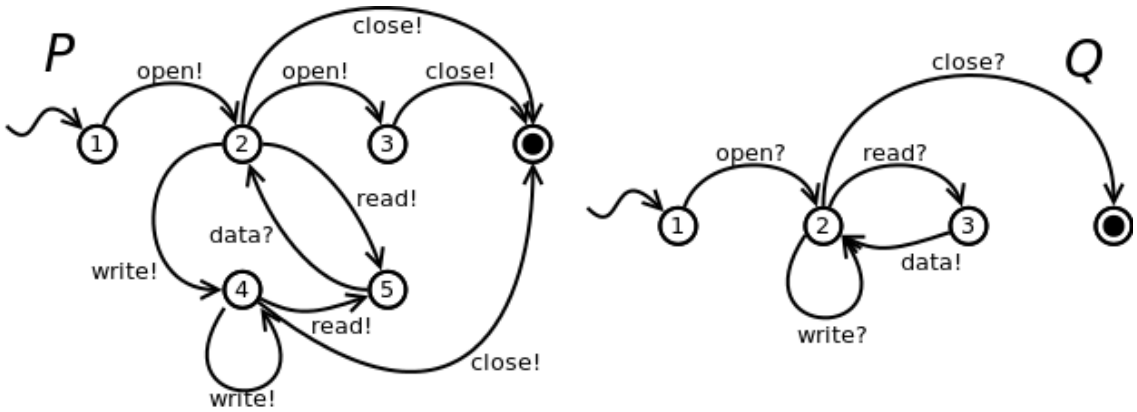


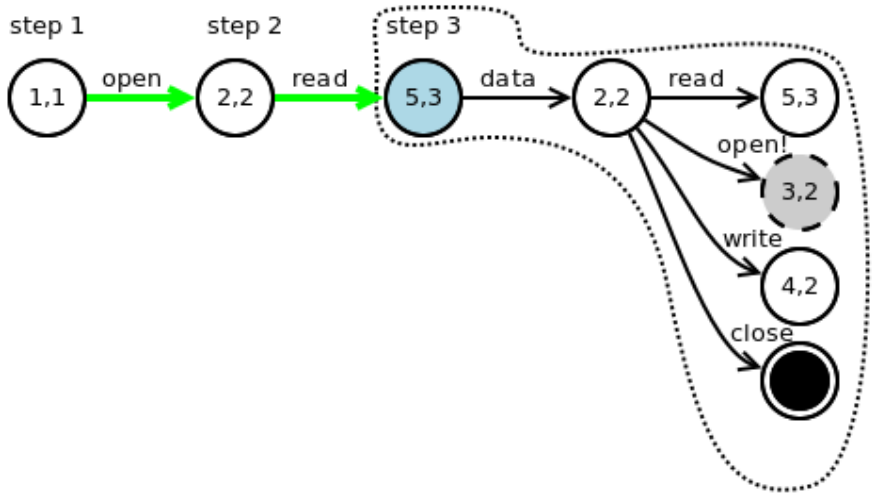
Sketching the idea ($l = 2$)



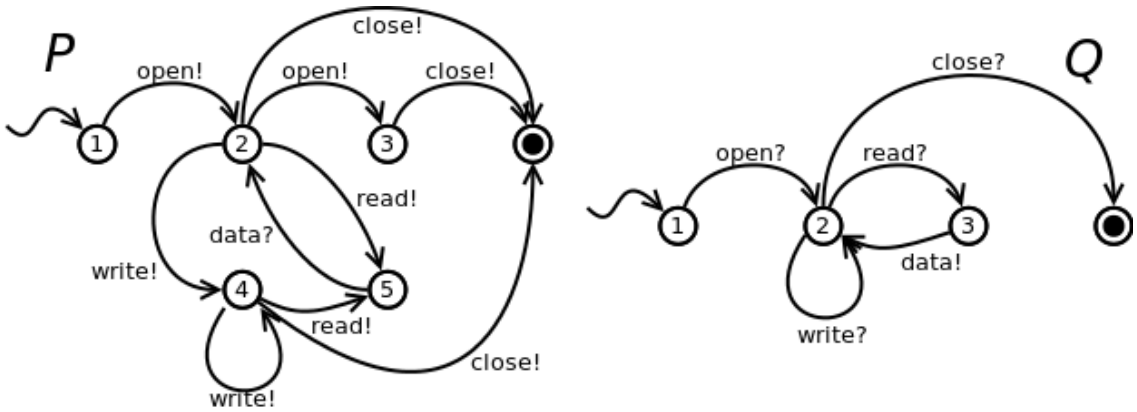


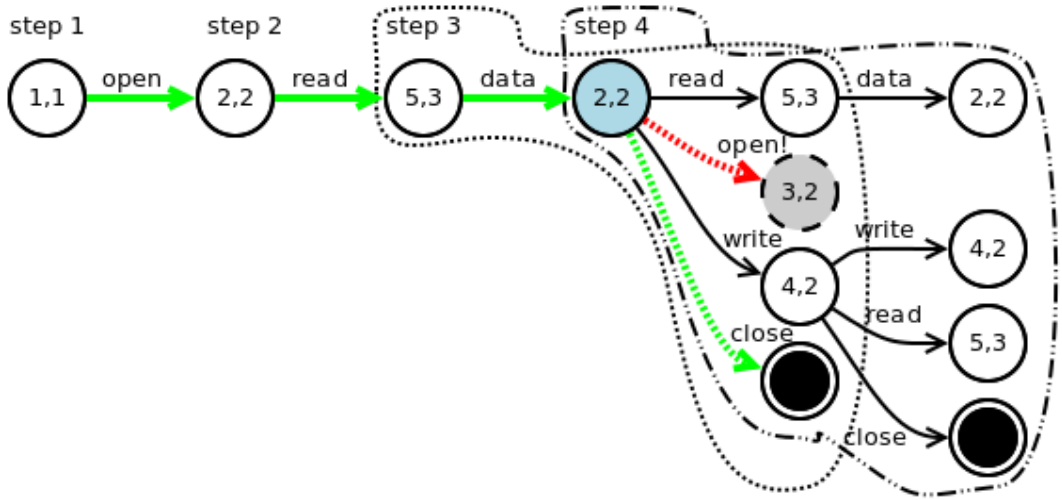
Sketching the idea ($l = 2$)





Sketching the idea ($l = 2$)





Cassandra - Concrete Implementation (on-going)

Cassandra has been implemented in the OSGi framework

OSGi

- Component based framework for Java
- components (bundle) can be remotely installed, started, stopped, updated and uninstalled
- Bundles should declare provided and required services in a manifesto (XML file)
- dependencies (optional and mandatory) are dynamically solved by the infrastructure

Cassandra in OSGi

- Cassandra is implemented as an OSGi bundle
- A system is the transitive closure of bundle dependencies
- All bundles are instrumented, via AOP techniques, to introduce a cassandra dependency
- Each method is instrumented with notification mechanisms to the Cassandra bundle

Conclusions and Future Work

To close started activities

1. Refine and Study complexities measures for algorithms
2. Finalize the implementation
3. Make experiments and evaluate possible overheads

Possible extensions

- Dynamic variation of the look-ahead value?
- Introduction of failure avoidance strategies to be combined with Cassandra
- ...

Thanks!!

??