FixD

Fault Detection, Bug Reporting, and Recoverability for Distributed Applications

David A. Noblet, Cristian Ţăpuş, and Jason Hickey
{dnoblet,crt,jyh}@caltech.edu

Mojave

Introduction

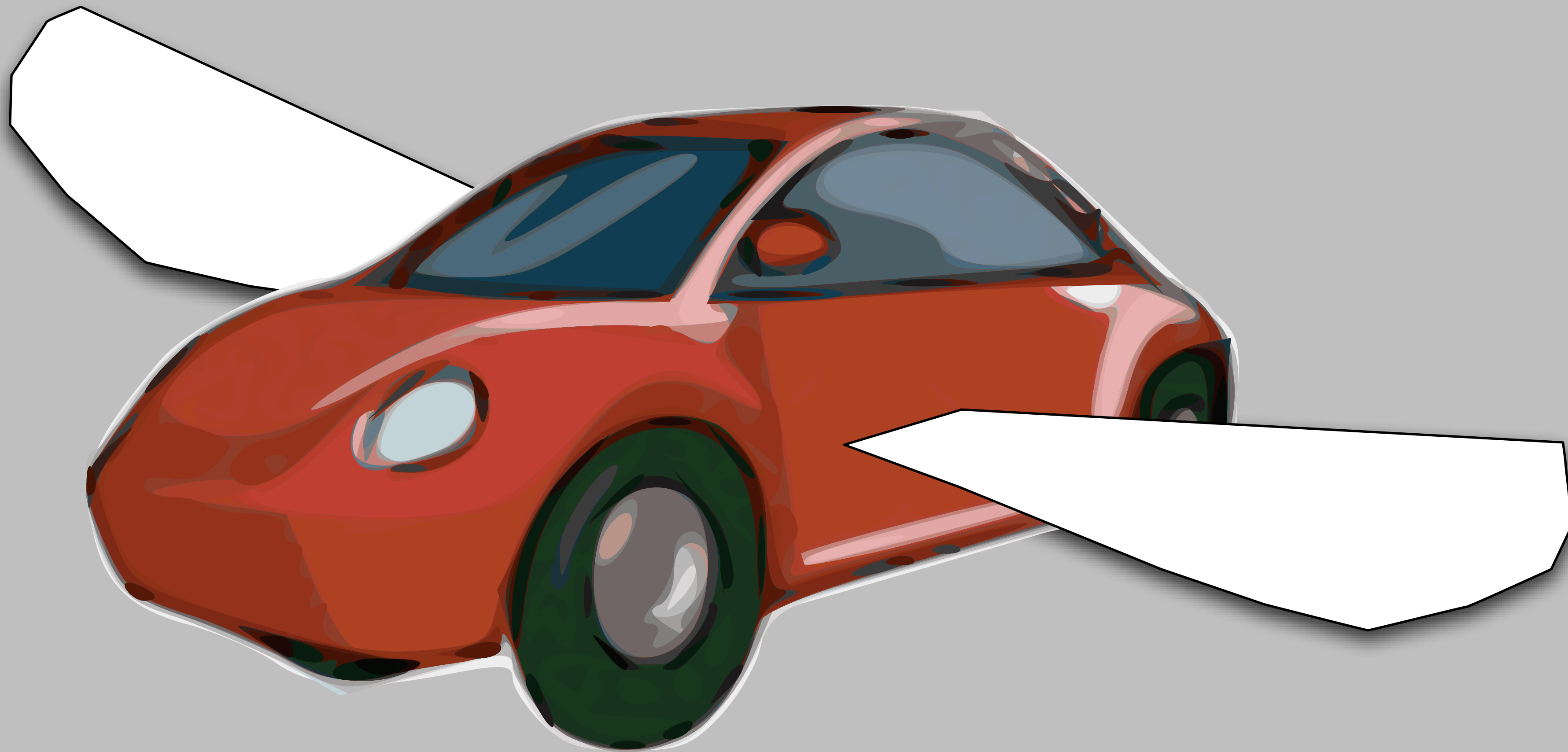
- Humans, in general, are really good at writing *bad* software
- As complexity of the systems we develop increases, this problem is only aggravated
  - Increasing concurrency & distribution of services
  - Heterogeneous components
- We need new techniques to increase reliability
  - Existing mechanisms are not enough

Mojave

A (somewhat silly) car analogy

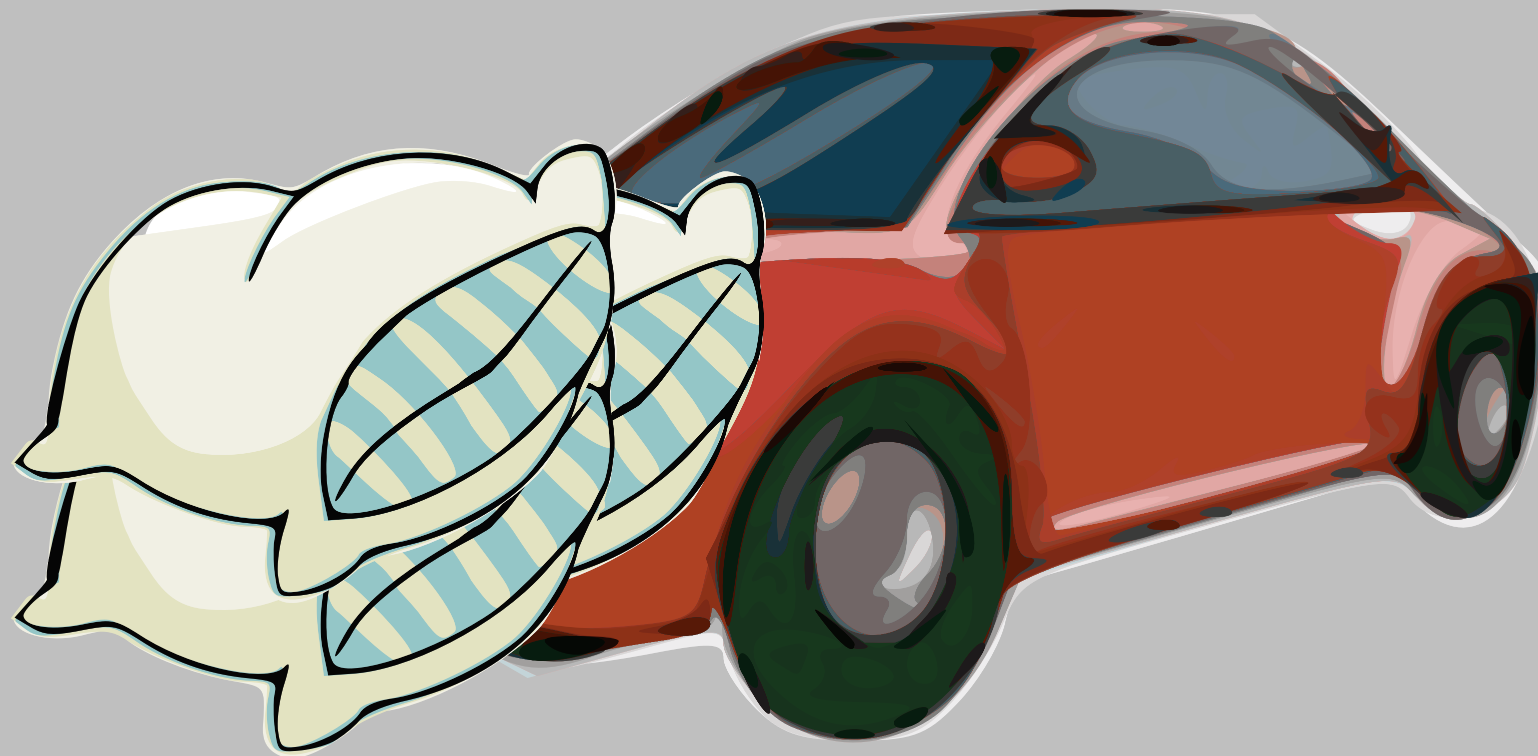
Mojave

Approach #1: Correctness by design



Mojave

Approach #2: Hardening (fault tolerance)



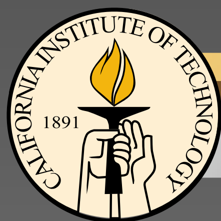
Mojave

What else can we do?

- Both of the previous approaches require *a priori* knowledge of either how *best* to:
  - Avoid error conditions in the first place
  - Recover from errors when they occur
- Instead, we can allow the agent to *introspectively* reason about its own behavior dynamically
  - Predict its future behavior
  - Dynamically avoid (predicted) errors

Mojave

Our approach: Model prediction



Mojave

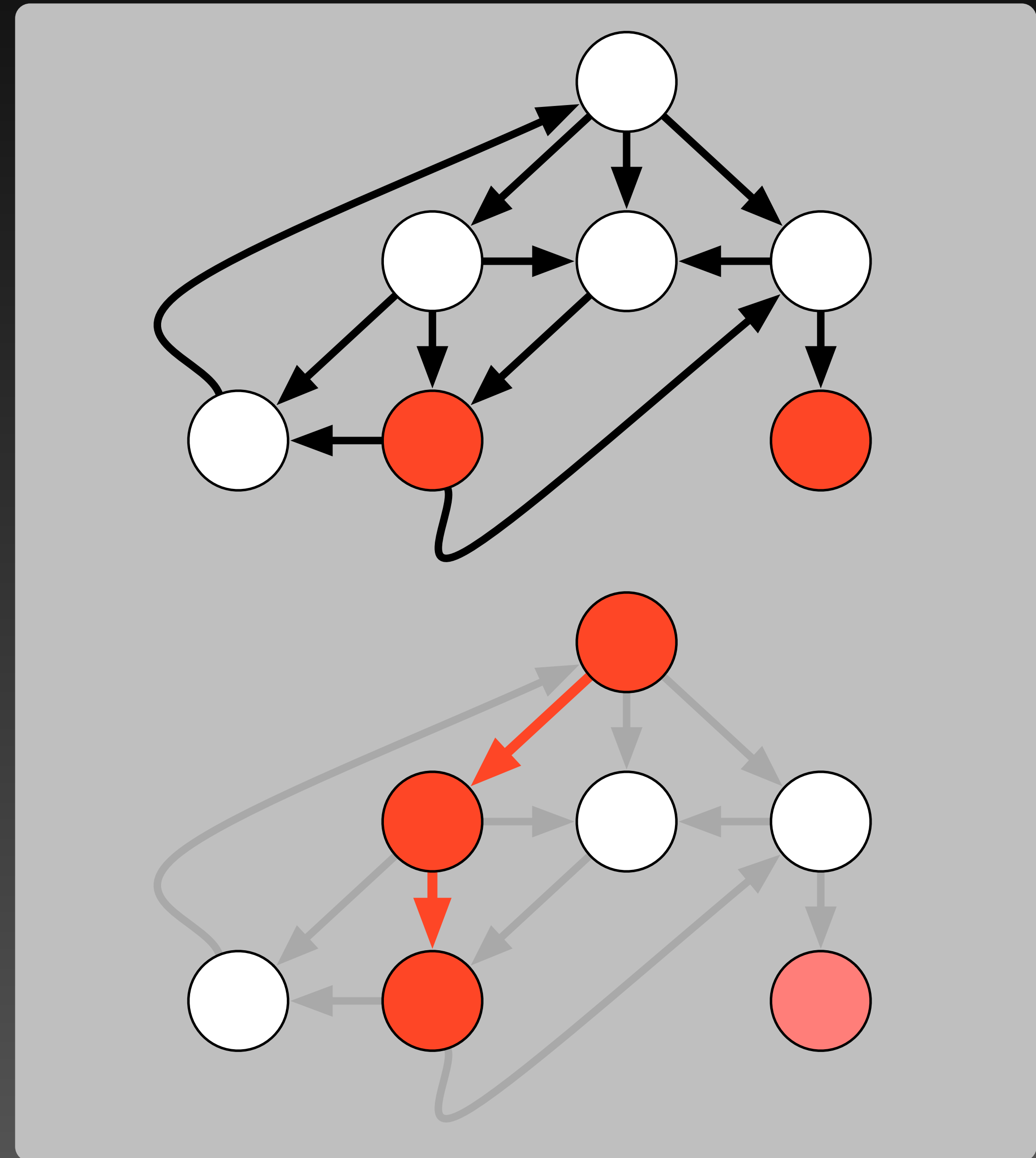
Back to software (no more cars, I promise!)

- So, what does this give us?
  - Simplified error recovery design
  - Potential for error avoidance in unexpected scenarios (i.e. those the the developer might not have foreseen)
- Fortunately, there are existing tools to reason about the dynamic behavior of a software system:
  - Model checking

Mojave

Model Checking (explicit state)

- A model checker is a tool designed to systematically explore the graph of reachable states as dictated by a model of the system being verified
- Users can specify the behavior of the system and properties to which it should adhere
- When a property is violated, the model checker responds with a counterexample that constitutes such a violation



CALIFORNIA INSTITUTE OF TECHNOLOGY 1891

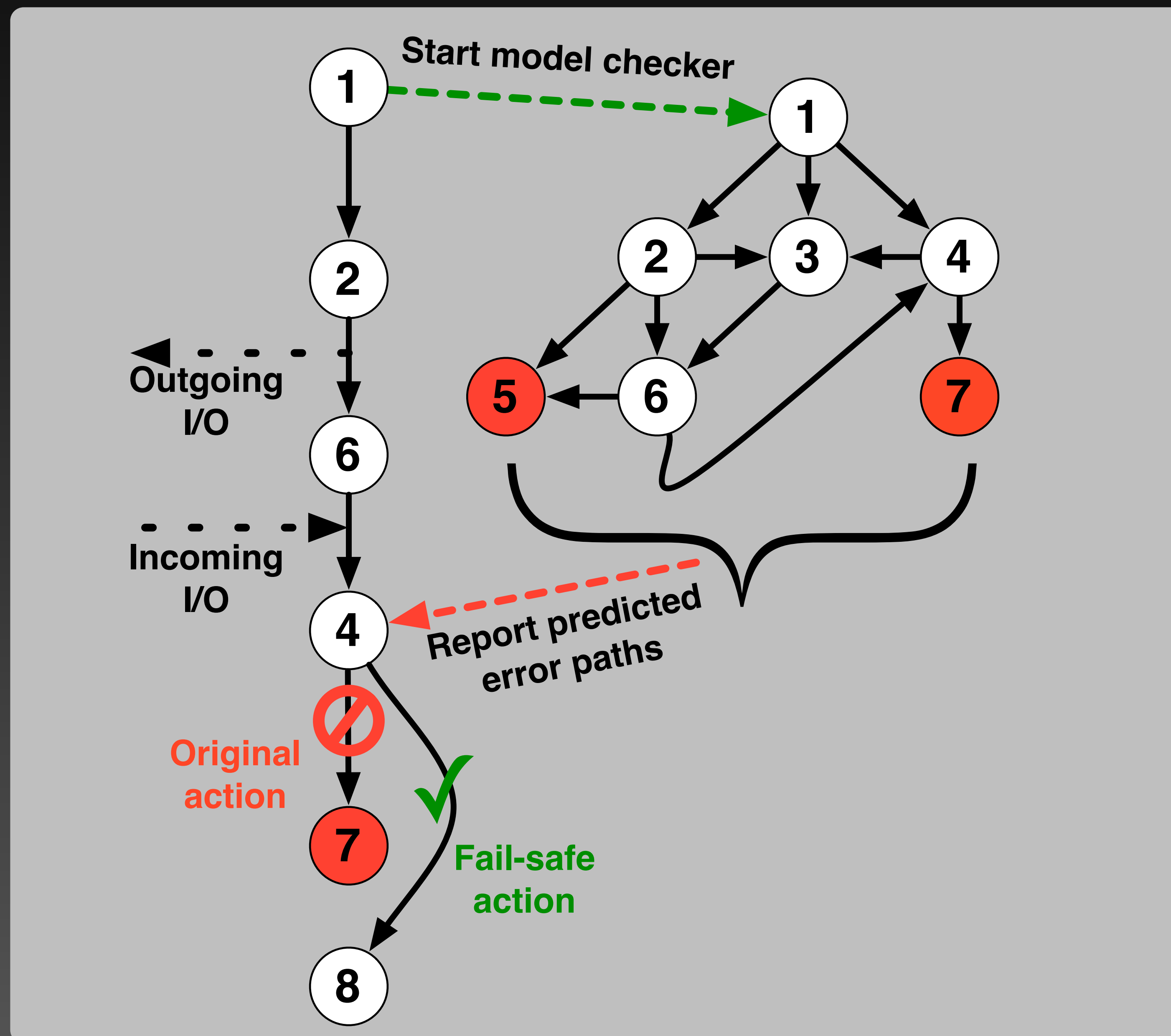
Mojave

FixD - Goals

- Design a new system, *FixD*
  - Facilitates dynamic recovery
  - Allows for absence of perfect global knowledge
  - Avoids centralized components
- Use a model checker to allow the application to predict its own behavior so that it can:
  - Identify potential erroneous execution paths
  - Avoid the faulty behavior before it ever occurs

Mojave

FixD - Example



Mojave

FixD - Overview & Design

- *FixD* allows a developer to write an application that is intended to run concurrently alongside a model of its behavior
- Periodically the application will start up the model checker and indicate for it to explore the state space of the system:
  - Starting from the current state of the application
  - Exploring only to a pre-determined (and finite) depth
- If the model checker discovers any errors:
  - It communicates the faulty execution paths to the application
  - And the application will try to avoid these paths

Mojave

FixD - Annotations

- We have designed *FixD* as a syntax extension (i.e. a set of annotations added) to the OCaml programming language
- When compiled, the *FixD* source is transformed into two components: the application itself, and its model
- Users annotate their OCaml programs to:
  - Facilitate model extraction
  - Identify properties (safety and liveness) of the code to which the system is supposed to adhere
  - Provide *fail-safe* procedures to replace the "normal" behavior of an application in the event that a regular procedure would violate one of the user-specified properties

Mojave

FixD - Graphical Overview

FixD source

FixD Front-end

OCaml source (model)

modelD Back-end

OCaml source (app.)

ocamlc

FixD Application

Mojave

Implementation Challenges

- Composing models
  - Other processes in the system & the environment
- Initialization of the model checker with a global state
  - Use previously explored states from the model checker (if it is fast enough!)
  - Or periodically take globally consistent snapshots
- Reconciling local execution paths (from the application) with global ones (returned by the model checker)

Mojave