A Top-down Approach to Managing Variability in Robotics Algorithms

Selma Kchir, Tewfik Ziadi, Mikal Ziane (LIP6, UPMC)
Serge Stinckwich (UMMISCO, UPMC/IRD)
DSLRob 2013, Tokyo

Variability in Robotics

- Robotics software must deal with different sources of variability:
  1. Different kinds of sensors, actuators,
  2. Different algorithms for the same task.
- Robotics has tried to address low-level variability.
- But the task is huge !
- Developing a generic obstacle avoider deemed daunting [Smart 2007] despite existing middleware.

Robotic Algorithms

- Robotics algorithms depend on low level details (sensors, actuators).
- Consequently they are:
  - difficult to understand,
  - difficult to adapt or combine,
  - impacted by changes in lower-level details.

Families of Algorithms

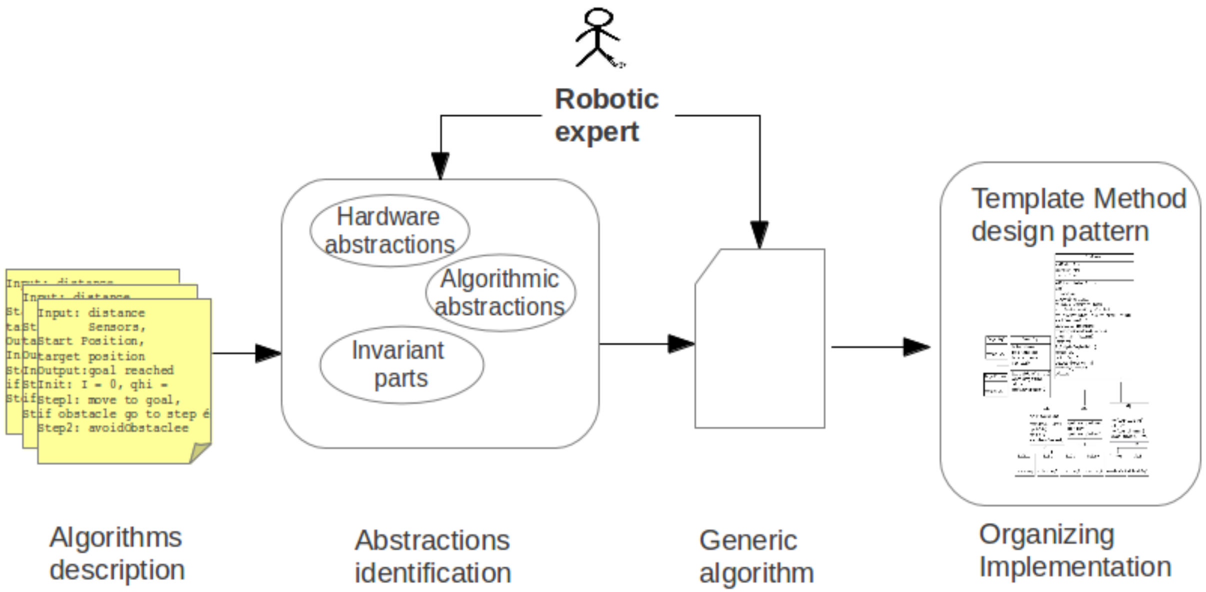
- Algorithms are often adapted: many variants are typically available.
- Algorithmic and low-level variability are usually intertwined!
- Hard to compare or even understand the variants.
- An approach is needed to organize families of algorithms:
  1. algorithmic choices must be clearly expressed
  2. and decoupled from implementation choices.

Expected Improvements

- A generic (common) algorithm is available and easily understood
- Variants can be compared and selected
- Algorithms are resistant to hardware changes
- New variants can more easily be introduced

Our Approach

- **Input:** a robotic task, a family of algorithms
- **Output:** generic algorithm and algorithmic, sensory or actions abstractions



Overview of the Approach

1. Define a generic algorithm expressed in terms of algorithmic sensory and action abstractions

2. Implement the algorithm

3. Implement or reuse the abstractions

   Abstractions are methods or sets of methods (type)

Generic Algorithm

- Common to all the variants
- Does not depend on low-level variants
- 2 kinds of hot-spots (variation points):
  - algorithmic
  - low-level (sensory or actuation)

Implementation

- Template Method design pattern
- Generic algorithm: an abstract class
  - concrete methods: fixed parts
  - polymorphic methods: variable parts
- Each variant of the algorithm is a subclass

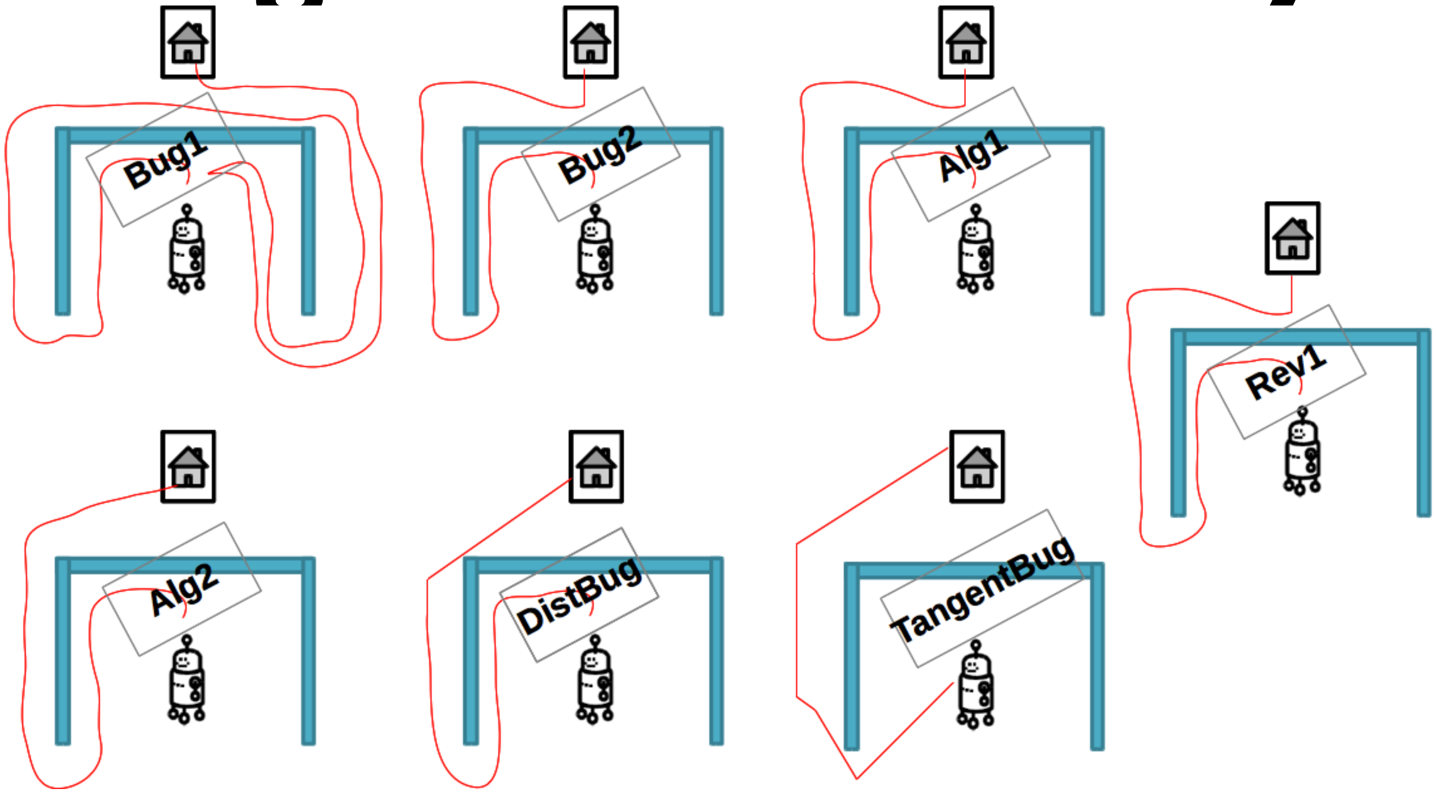
Actuation and Sensing

- Use delegation to avoid combinatorial explosion
  - separate different concerns (sense, plan, act)
  - ➡ actuation, sensory and algorithms are different hierarchies
- Many ways to organize the hierarchies => global effort of the community
- Different from middleware
  - much higher-level
  - abstractions defined top-down rather than bottom-up

Application to Bug Algorithms Family

- Bug navigation family:
    - the robot moves from a start point to the goal
    - the robot must avoid the obstacles on its way to the goal
- Bug navigation assumptions:
    - two dimensional unknown environments
    - the robot is considered as a point
    - the robot has perfect sensors and a perfect localization ability

Application to Bug Algorithms Family



Bug algorithms are about 20 variants among them 7 variants are considered here

Bug Algorithms: Description

(1) The robot ***moves its goal*** until an ***obstacle is detected*** on its way.

(2) From the point where the obstacle were encountered (hit point), the robot ***looks for a point*** (leave point) ***around the encountered obstacle*** to move to its goal again.

(3) Once the ***leave point is identified***, the robot **moves to it** and leaves the obstacle.

Step (1), (2), (3) are repeated until the goal is reached or the goal is unreachable.

Bug Algorithms: Abstraction Identification

- Hardware abstractions: obstacle detection, localization
  - **getPosition()**
  - **getSafeDistance()**
  - **obstacleInFrontOfTheRobot()**
- Algorithmic abstractions: mostly related to the leave point
  - **findLeavepoint(Point robotPos, Point hitPoint, Point goalPos)**
  - **identifyLeavePoint(bool direction, Point robotPos, Point goalPos)**
  - **bool leavePointFound()**

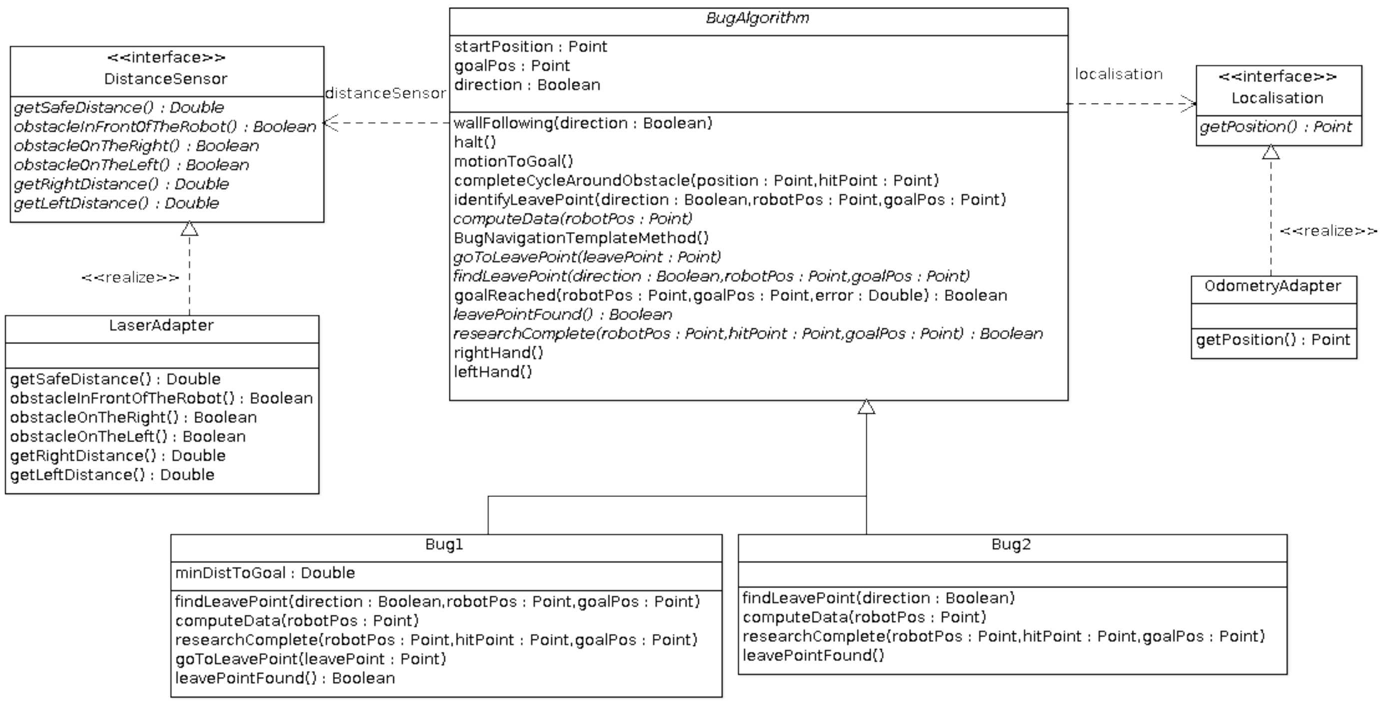
Bug Algorithms generic Algorithm

Sensors : A perfect localization method.
An obstacle detection sensor
input : Position of Start (q_{start}), Position of Target (q_{goal})
Initialisation: robotPos ← `getPosition();`
direction ← `getDirection();`

```
if goalReached(robotPos) then
  | EXIT_SUCCESS;
end
else if obstacleInFrontOfTheRobot() == true
then
  | identifyLeavePoint (direction, robotPos,
  | goalPos);
  | if leavePointFound() &&
  | researchComplete(robotPos, getHitPoint(),
  | goalPosition) then
  |   | goToLeavePoint(getLeavePoint());
  |   | faceGoal()();
  | end
  | else if
  | completeCycleAroundObstacle(robotPos,
  | getHitPoint()) && !leavePointFound() then
  |   | EXIT_FAILURE;
  | end
end
```

Organizing Implementation

- Middlewares: OROCOS-RTT+ROS
- Simulation: Stage-ROS
- https://github.com/SelmaKchir/BugAlgorithms



Conclusions

- A top-down approach to manage variability in a family of algorithms:
  - Step 1: Define generic algorithm and abstractions
  - Step 2: Implement
- Validation on 7 variants of the Bug Family

Perspectives

- Consider software product lines if many variants

Problem: still unclear how to best define algorithmic product lines

- Libraries of abstractions

to complement middleware implementations

- Defining a (reasonably) generic obstacle avoider would not be so daunting any more.