Adaptive Deployment of Safety Monitors of Autonomous Systems

4. Implementation:

4.1 Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSP) are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations [1]. Constraint refers to rule, limitation or restriction. CSP describes an efficient way to solve a wide variety of problems. A **factored representation** is used for each state which is represented as a set of variables, each of which has a value. The problem is solved when each variable has a value that satisfies all the constraints on the variable. [2]

A constraint satisfaction problem consists of the set of variables (X), domain for each variable (D), and set of constraints (C). It can be formulated as follows:

X is a set of variables, {X_1, . . . ,X_n}.
D is a set of domains, {D_1, . . . ,D_n}.
C is a set of constraints that specify allowable combinations of values.

Each domain consists of a set of allowable values {v_1, . . . , v_k} for variable X_i. For example, A1 and A2 are the two variables with domain {X,Y}, then the constraint stating the two variables must have different values can be defined as C={A1≠A2} [2]

The values will be assigned to some or all of the variables by means of assignment. An assignment is said to be complete when each and every variable is assigned with a value. A solution obtained by solving CSP will be an assignment that is complete satisfying all the constraints.

Advantages:

- CSP can keep track of the variables violating a constraint.
- It helps in clean specification of many problems, generic goal, successor function and heuristic. When a problem is represented as a CSP, it can be solved with general package.
- All the branches that violate constraints will be pruned off automatically.
- CSP is faster than state space searchers because the large swatches of the search space can be eliminated quickly. The problems that are hard to control for regular state space search can be solved quickly when formulated as a CSP

4.2 MiniZinc

MiniZinc is a free and open source constraint modeling language. It can be used to model constraint satisfaction and optimization problems as a higher level. It is a language designed where constrained optimization and decision problems can be specified over integers and real numbers. It is designed to act as an interface between different backend solvers easily. An input Minizinc (.mzn) model and data file is transformed into a FlatZinc model. FlatZinc model consist of variable declaration along with constraint definitions and objective function definition for optimization problem. The translation from MiniZinc to FlatZinc is performed to individual backend solvers, so that the constraints will be controlled. Particularly, MiniZinc allows the specification of global constraints by means of decomposition. [3]

4.2.1 Minizinc as an IDE:

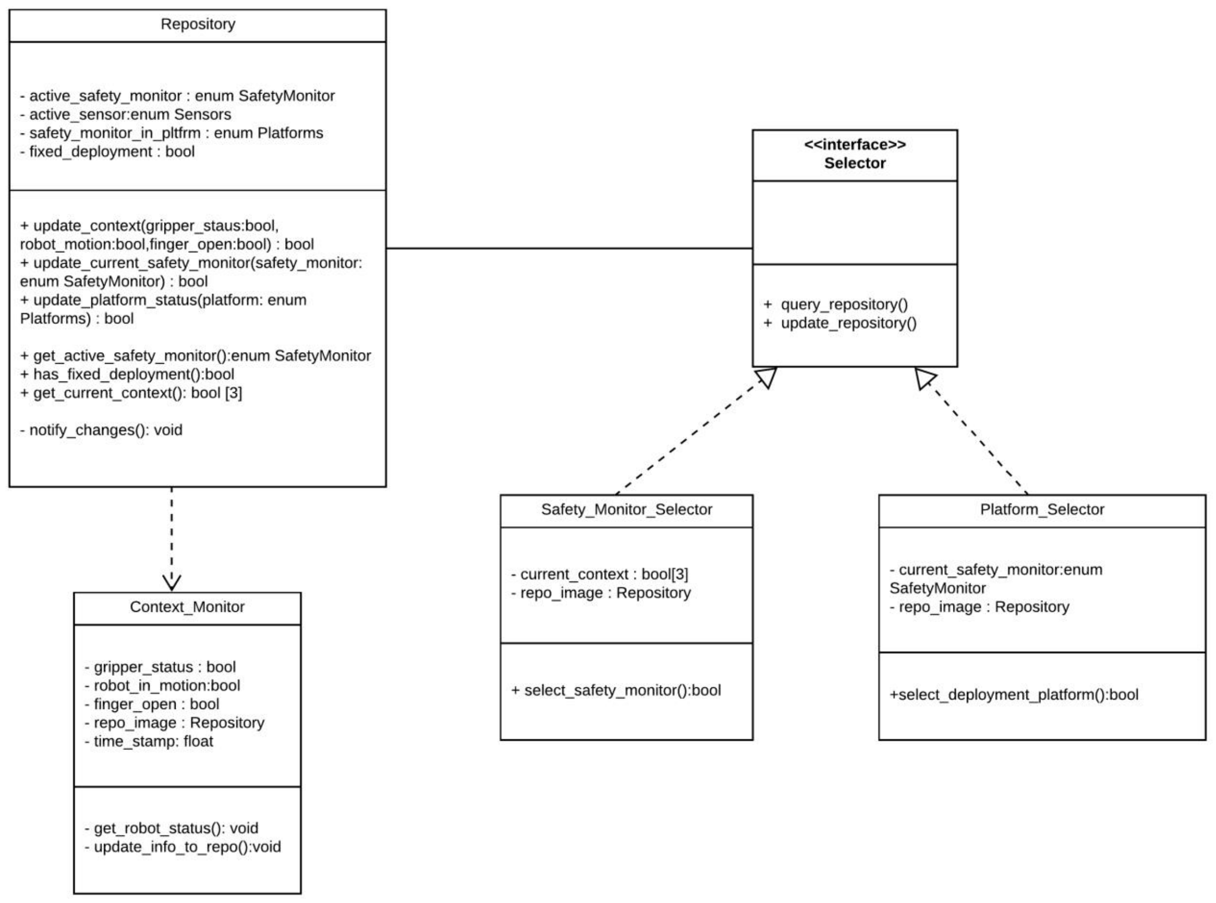
MiniZinc comes with an Integrated Development Environment (IDE), that makes it easier for developing and executing constraint models. It is user friendly. IDE facilitates in writing and running MiniZinc models. It is provided with tabbed editor with highlighter for MiniZinc syntax followed by configuration dialogs for solver options and model parameters, and an integrated environment where the models can be compiled and running solvers. [4]

MiniZinc IDE’s latest version is 2.4.3 which was released on 2020. It can be installed as bundled package that consists of Minizinc library, IDE, solvers and interfaces. Minizinc is used to solve the CSP and return the solutions satisfying given constraints.

4.2.2 Minizinc Python API:

It is a python package that allows accessing all of MiniZinc's functionalities directly from Python. It provides an interface from Python to MiniZinc driver. It provides easy access to MiniZinc using native Python structures. This allows more scripts to run MiniZinc, but will also allow the integration of MiniZinc models within bigger python projects. It can be installed using pip command. [5]

4.3 Class Diagram


Repository
- active_safety_monitor : enum SafetyMonitor
- active_sensor:enum Sensors
- safety_monitor_in_pltfrm : enum Platforms
- fixed_deployment : bool
+ update_context(gripper_staus:bool, robot_motion:bool,finger_open:bool) : bool
+ update_current_safety_monitor(safety_monitor: enum SafetyMonitor) : bool
+ update_platform_status(platform: enum Platforms) : bool
+ get_active_safety_monitor():enum SafetyMonitor
+ has_fixed_deployment():bool
+ get_current_context(): bool [3]
- notify_changes(): void
<<interface>> Selector
+ query_repository()
+ update_repository()
Context_Monitor
- gripper_status : bool
- robot_in_motion:bool
- finger_open : bool
- repo_image : Repository
- time_stamp: float
- get_robot_status(): void
- update_info_to_repo():void
Safety_Monitor_Selector
- current_context : bool[3]
- repo_image : Repository
+ select_safety_monitor():bool
Platform_Selector
- current_safety_monitor:enum SafetyMonitor
- repo_image : Repository
+select_deployment_platform():bool

The class context monitor provides the contextual information needed for selection and deployment. It consists of attributes namely robot in motion and gripper status where the values are fetched from the environment by the robot (i.e.) values are updated dynamically at every time interval t. The state of the robot is sent to the repository and updated. Repository contains information for deployment. It is the central component in the architecture. Every component updates the information to the repository. All the components are connected to the repository. The repository class provides attributes for storing the values of robot's current context, selected safety monitor and platform for deployment.

Selector is an interface that acts as a communicator between repository class and other two classes namely safety monitor selector, platform selector. The selector interface provides two methods for querying and updating information to the repository. Safety monitor selector class contains definition for selecting the safety monitor based on the current context. The selected safety monitor information is updated in the repository using interface. The safety monitor class selects a suitable safety monitor using select_safety_monitor method by checking the current context of the robot and updates the repository accordingly. The platform selector class selects the suitable platform based on the selected safety monitor. Even the selected platform information is updated to the repository. The platform that satisfies the given requirements will be selected. This is facilitated by formulating the problem as CSP and solved using MiniZinc. The requirements are formulated into suitable constraints and platforms that satisfy the given constraints will be selected. Minizinc contains definitions for the formulated problem along with the constraints.

The design pattern followed is interface pattern. The main idea is the functions are separated from implementations. It can be used when it is necessary to know how the messages are exchanged within classes (i.e.) when a class should be reused each and every time, the communication interface will be declared as an Interface type. [6]

References:

[1] https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

[2] Artificial Intelligence: A Modern Approach by Peter Norvig and Stuart J. Russell

[3] https://www.minizinc.org/tutorial/minizinc-tute.pdf

[4] https://www.minizinc.org/ide/

[5] https://pypi.org/project/minizinc/

[6] http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/interface.html