2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Madrid, Spain, October 1-5, 2018

Move Base Flex
A Highly Flexible Navigation Framework for Mobile Robots

Sebastian Pütz[1], Jorge Santos Simón[2], and Joachim Hertzberg[1,3]

***Abstract*— We present Move Base Flex (MBF), a highly flexible, modular, map-independent, open-source navigation framework for use in ROS. MBF provides modular actions for executing plugins for path planning, motion control, and recovery. These actions define interfaces for external executives to allow highly flexible navigation strategies, which can be intertwined with other robot tasks. MBF has been successfully deployed in a professional setting at customer facilities to control robots in highly dynamic environments. We compare MBF with the well-known `move_base` and present the architecture as well as different deployment approaches, including how MBF can be used with different executives to perform complex navigation tasks interleaved with other robot operations.**

I. Introduction

In robotics, autonomous high-level navigation tasks are increasingly interdependent with other high-level operations. Commonly, robot navigation tasks are embedded in complex applications, e.g., fetch-deliver scenarios, guided tours, rescue operations, and exploration of unknown environments. However, different terrains, e.g., indoor vs. outdoor, require different navigation techniques and usually different map representations. Moreover, the abilities, characteristics and limitations particular to any robot further constrain the choice of map representations, planners and controllers.

However, almost always one or more planners are necessary to compute a collision-free path, and one or more controllers are used to compute the velocity commands to follow that path, while avoiding unmapped obstacles. And most surely, some kind of strategies are needed to allow the robot to recover from the most common failures during navigation.

By designing *Move Base Flex* (MBF), we attempted to meet the described challenges without sacrificing flexibility. We present MBF, a highly modular and flexible navigation framework, conceived as a replacement for `move_base` [1], the central module of the navigation stack in ROS, the *Robot Operating System* [2], [3]. Section II places MBF in the context of related research and discusses state-of-the-art software approaches.

Dealing not only with the current shortcomings of robot navigation, but also with the integration into high-level applications, MBF is of great use for companies, like *Magazino*[2], running fleets of robots to reliably navigate in very large warehouses.

Section III presents the architecture and modular expandability of MBF, its backwards-compatibility with current ROS navigation, and how to load and switch plugins at runtime, while providing actions and services as interfaces for external executives. We describe the abstract navigation core, which provides a map-independent execution and interface logic.

In section IV we provide two deployment scenarios, one using `SMACH` [4], and another using a *Behavior Tree* developed by *Magazino*. MBF was successfully deployed at customer facilities to control *TORU* robots in highly dynamic environments. The MBF sources and examples used later in this paper are publicly available.

II. Related Work

A. Robot Operating System

More and more companies and research institutes use ROS, such as *Fetch Robotics*, *Clearpath Robotics*, *NASA*, *PAL Robotics*, *Yujin Robot*, *Intel*, *UC Berkely*, *Oregon State University*, *Bosch Research and Technology Center*, *ETH Zurich*, *Osnabrück University*, etc. [3].

Before ROS became an ecosystem, several research robotics software frameworks – among them the *Stanford Artificial Intelligence Robot* (STAIR) and the *Personal Robot* (PR2) program – were prototyped and developed. After 2007, *Willow Garage* were formed by the *Stanford University* programs and provided well-tested implementations[2], [3]. In the last decade, ROS has established itself as the leading open source robotics platform, supported and used by numerous research institutes and companies around the world, who contribute by developing and extending the ROS core, as well as an ever increasing number of ROS packages.

Since a couple of years the *Open Source Robotics Foundation* took over the maintenance of the ROS infrastructure. Worldwide, tens of thousands of users, from very diverse fields, use ROS to program their systems and robots, from small personal projects to large industrial intelligent systems.

B. Navigation Planning

The ROS navigation stack has been widely used in combination with `move_base`, layered costmaps [5], *OpenSlam's GMapping* based on improved techniques for grid mapping [6], and robot pose EKF.

Marder-Eppstein et al. [7] describe the combination of navigation components for robust and successful robot navigation in an office environment. Planning is divided into

[1]Knowledge Based Systems Group, Institute of Computer Science, Osnabrück University, Wachsbleiche 27, 49090 Osnabrück, Germany `spuetz@uni-osnabrueck.de`
[2]Magazino GmbH, Landsberger Str. 234, 80687 München, Germany `santos@magazino.eu`
[3]DFKI Robotics Innovation Center, Osnabrück branch, Albert-Einstein-Str. 1, 49076 Osnabrück, Germany `joachim.hertzberg@dfki.de`

a global and a local stage by using an underlying global costmap to represent the static map and a local costmap to represent the local environment, storing dynamic obstacle data detected by range sensors. A global planner bound to the global costmap is used as a high-level guide for path planning from a start pose (usually the robot pose) to the goal pose. The global costmap usually represents the already mapped environment.

As described in [7], recovery behaviors should make the system robust while taking the local and the global costmap into account. A number of these behaviors are called in a fixed sequence when the robot gets stuck. We have observed that the fixed sequence limits the flexibility and costs time in the navigation process, due to going through useless recovery strategies regarding the current problem as shown in Section V.

Plugin interfaces for local planners, global planners, and recovery behaviors are defined in ROS's `nav_core` package. Thus, many developers have written their own planner, controller, and recovery behavior plugins following the `nav_core` API.

The `move_base` ROS package connects the global and local planner, the recovery behaviors, and the global and local costmap. When the program starts, the global planner, local planner, and recovery behavior plugins are loaded and instanciated with pointers to costmaps and pointers to the transformation system *tf* [2]. `move_base` is implemented as a hard-coded finite state machine (FSM) handling the full sequence of navigation: planning, control, and recovery. The hard-coded FSM restricts the flexibility and allows no situation-dependent handling and execution of specific behaviors.

C. Flexible Navigation

Many modern robot systems use hybrid architectures to accomplish their goals. Personal robotics applications consist of hundreds of components, which need to be integrated [4]. As described by Conner et al. [8], hierarchical finite state machines (HFSM) are often used for integrating all system parts into a hybrid architecture that defines the overall system behavior. In an HFSM (also known as State Charts [9]), a state can contain one or more sub-states. Colledanchise and Ögren [10] list these different architecture types, such as subsumption architectures, teleo-reactive programs, decision trees and behavior trees (BT), describe their advantages and disadvantages, and argue for BTs. Boren and Cousins [4] introduced SMACH, which is based on the concepts of hierarchical concurrent state machines(HCSM). While the SMACH library is ROS-independent, they provide a `smach_ros` package for communicating with the ROS system, including topics, services, and actions.

Conner et al. [8] presented an implementation of a flexible navigation system, providing separate executables and actions for path planning and motion control, e.g., *GetPath*, *FollowPath*, *FollowTopic*, *ClearCostmap*. In each executable, another instance of a ROS *costmap_2d* is used to provide maps for local and global planners. This leads to an expensive computational overhead when managing the costmap instances in multiple executables. Moreover, recovery behavior plugins, which in `move_base` shared an instance with both the local planner and the global planner, cannot be loaded. Conner et al. bring forward the argument that recovery behaviors can be customized using additional plugins. MBF supports recovery behavior plugins to keep the backwards-compatibility. However, we identified costmap (`costmap_2d`) instances as a quite computationally expensive resource.

Conner et al. use *FlexBE* [11] to integrate their navigation system into a high-level HFSM. FlexBe is an extension of SMACH. In general, their flexible navigation system allows to create navigation tasks in the manner of an action and operation selection, decomposing the navigation task into path planning and motion control, while allowing to interrupt the execution and to re-plan if necessary.

Move Base Flex, as presented next, is a system dealing with the mentioned shortcomings. It features a map-independent modular core implementation, which uses enhanced plugin and action interfaces. To avoid expensive computational costs of redundant *costmap_2d* instances – when using a complete navigation system with planning, control and recovery – the MBF navigation server shares these instances with all loaded plugins.

III. Architecture & Actions

MBF's system design addresses several constraints to make it as flexible and modular as possible and to deal with the shortcomings of common navigation frameworks just described, at the same time addressing navigation needs that are but one aspect of an overall complex robot task.

A. Abstraction for Map Independence

In developing MBF, we targeted a map-independent navigation framework. For example, quite a few navigation systems, indoors or outdoors, require 3D map representations, and so 2D costmaps are no longer appropriate. To achieve map-independence, we split the framework into an abstract MBF level and a map implementation level. The abstract level provides the core functionality without a connection to any map instance. The map implementation level inherits the abstract level, connects the plugins with the corresponding map instance, and provides additional map-specific functionalities and interfaces.

Figure 1 sketches the MBF architecture. The *Abstract Move Base Flex Level* is represented by a package called `mbf_abstract_nav`. The *Costmap Implementation Level* shows the inheritance of the abstract level together with the connection of the global and the local costmap to MBF. The lightweight costmap implementation level is implemented in the `mbf_costmap_nav` package. It only manages the instantiation of the costmaps and handles the loading and initialization of the extended abstract plugin interface, passing a costmap pointer to instantiated plugins. Beside using costmaps, MBF has been tested with a mesh navigation planner and controller for outdoor rough terrain navigation [12].

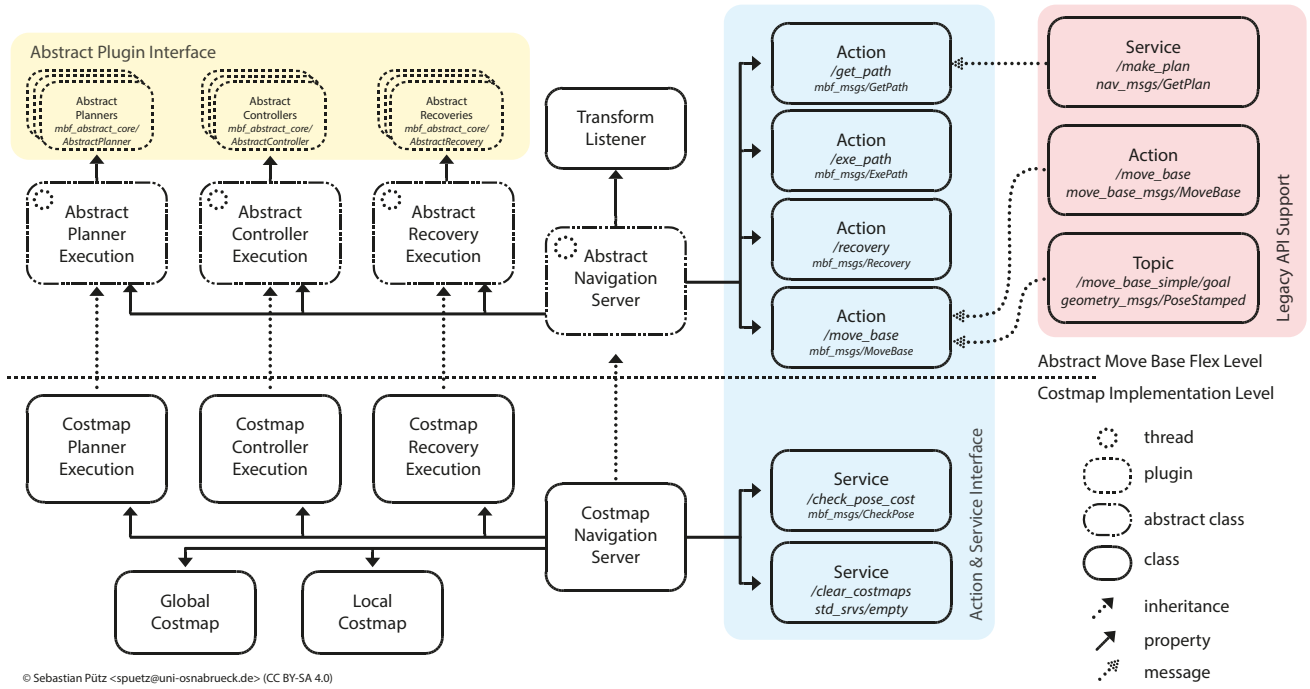

Fig. 1: The MBF architecture. Shown are the *Abstract Move Base Flex Level* and the *Costmap Implementation Level*; The *Plugin Interface* is highlighted in yellow; the *Action & Service Interface* for the external executive in blue; and the *Legacy API Support* in red.

Additionally, navigation servers providing the plugins with access to other map representations are easy to develop, because of their lightweight structure and the inheritance of the `mbf_abstract_nav` and `mbf_abstract_core` package.

B. Actions as External Executive Interfaces

The *Abstract Navigation Server* owns action servers using the ROS `actionlib` to provide the asynchronous actions *get_path*, *exe_path*, *recovery*, and *move_base*, which are defined in the `mbf_msgs` package. Each action returns, among others, an *outcome* number code. The *outcome*, for instance, can be used for a more specific error handling by the external executive. Developers of the external executive can use `outcome` codes defined in the plugins, as well as predefined codes from the action definition files, e.g., failure, canceled, invalid start or goal, no path found, patience exceeded, empty path, tf error, not initialized, invalid plugin, internal error, stopped, etc. The actions provided by MBF are detailed next.

1) `get_path`: This action computes a path from a given start pose or the current robot pose to a given target pose. The user may define which planner should be used and if the planner should use a tolerance due to an obstructed goal. The action returns the computed path, a corresponding cost, `outcome`, and message.

2) `exe_path`: Given a path and a controller name as input, this action hands over the path to the selected controller. It starts the selected controller and executes it in predefined frequency, while computing velocity commands, which are directly published to command the robot to move. During its execution it returns the current robot pose, velocity, and the distance and angle to the target. As a result, it returns the robot's final pose and the distance and angle to the target, as well as an *outcome* and a corresponding message.

3) `recovery`: The recovery action just executes a specified recovery strategy and returns an *outcome* with a corresponding message.

4) `move_base`: To provide a simple way of executing a navigation task like "go from A to B". MBF provides the `move_base` action which does it all together: planning, control and recovery if necessary, like it is done in the ROS `move_base` package. As input it takes a target pose, a controller name, a planner name, and a list of recoveries. It provides the same feedback and result as `exe_path`.

C. Planner, Controller & Recovery Interfaces

In the current ROS navigation stack, the `nav_core` plugin interfaces are named: *BaseGlobalPlanner*, *BaseLocalPlanner*, and *RecoveryBehavior*. We named the interfaces: *AbstractPlanner*, *AbstractController*, and *AbstractRecovery* to restore a consistent naming. The interface classes are defined in the `mbf_abstract_core` package. As mentioned above, plugin developers can specify an `outcome` code and a message, which can be used by the action server. The abstract interfaces refer to no map representation, e.g., given through an initialize method, as in the `nav_core`. References to the map representations are made by derived interface classes. For instance, derived interfaces providing an initialize method with a pointer to a costmap are defined in the `mbf_costmap_core` package: *CostmapPlanner*, *CostmapController*, and *CostmapRecovery*. The Plugins

(a) Toru

(b) Soto

Fig. 2: *Magazino's* robots

(a) Pluto

(b) Ceres

Fig. 3: Osnabrück University robots

have to be loaded and initialized in the derived navigation package. The `mbf_costmap_nav` is a derived package of `mbf_abstract_nav` and overwrites loading and initialization methods, which use the interface class definitions and the `pluginlib` defined in the `mbf_costmap_core` package, which is derived from `mbf_abstract_core`.

D. Backwards Compatibility

We decided to support the already existing global planners, local planners, and recovery behaviors for `move_base`, to open up MBF to the open-source community and simplify the getting-started and first-steps. Additionally, we designed the plugin interfaces in a way that plugin developers can make their plugins usable for both `move_base` and MBF at the same time. Thus an interchange of `move_base` and MBF is straightforward.

The backwards compatibility is achieved by loading and instantiating wrapper plugins as a fallback supporting all plugins, which implements a `nav_core` interface.

E. Multiple Planners and Controller

MBF provides the possibility to load multiple controller, planner, and recovery plugins. Listing 1 shows an example config file. It configures two local planner plugins – compatible with the `nav_core` – by defining a list of name and type pairs. The type defines the exported plugin. The name is chosen by the user and is used for selecting a specific planner, controller, or recovery behavior at runtime. Furthermore, the name is used as an MBF sub-namespace for all parameters of the plugin. Thereby, the same plugin may be loaded multiple times with different configurations. Thus, an external executive can choose a specific planner, controller or recovery plugin for different scenarios or environments, e.g., motion control in free space vs. line following.

```
controllers:
  - name: 'trajectory_planner'
    type: 'base_local_planner/TrajectoryPlannerROS'
  - name: 'teb_planner'
    type: 'teb_local_planner/TebLocalPlannerROS'

trajectory_planner:
  {trajectory_planner parameters}
teb_planner:
  {teb_planner parameters}
```

Listing 1: Loads multiple controllers, which are available under a user defined name, e.g., *trajectory_planner* and *teb_planner*

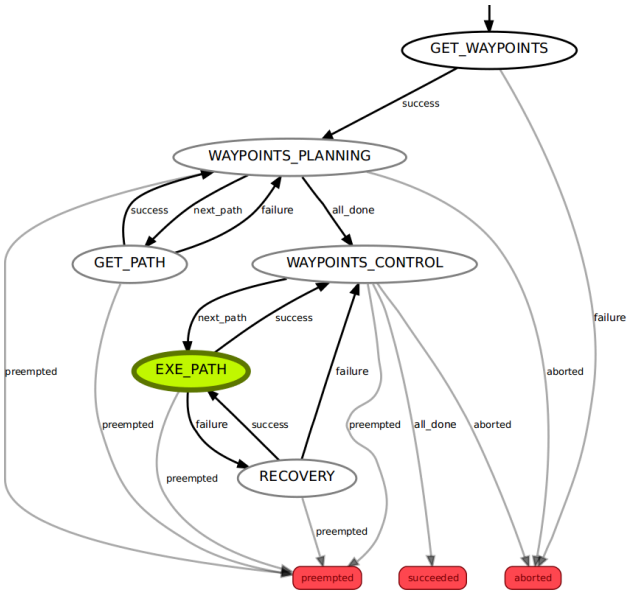

Fig. 4: A SMACH performing waypoint planning and motion control using the actions *get_path*, *exe_path*, and *recovery*.

IV. Deployment & Applications

MBF is currently used by Magazino[1], *Osnabrück University*[2], *Ruvu*[3], and TU Eindhoven robotics lab. Ruvu and TU Eindhoven robotics use MBF in combination with FlexBe [11].

Magazino use a flexible strategy represented as a behavior tree (BT) for its robots. MBF runs on *Magazino*'s robots (Fig. 2) for more than a year, in six different commercial settings, each making an average of 1.3 km per working day.

The research at *Osnabrück University* includes map-independent flexible navigation, high-level task planning, and semantic mapping with different map representations. In many of the respective robots (Fig. 3), a SMACH state machine is used in combination with MBF to connect the mentioned research fields.

Move Base Flex is open-source available at GitHub[4]. The documentation and tutorials – e.g. how to write a MBF plugin, how to use MBF with SMACH or a BT – can be found in the ROS Wiki[5].

Next we present a SMACH example and how *Magazino* uses a BT implementation with MBF.

A. SMACH

Figure 4 demonstrates an example SMACH for waypoint navigation. 1.) It performs a successful waypoint planning using *get_path*. 2.) It selects promising paths to minimize the total distance. 3.) It controls *Ceres*, shown in Fig. 3(b), using the already computed paths and hands them over step by step to the *exe_path* action and calls a *recovery* if necessary. A demo is open-source available in combination with a ready-to-start Gazebo simulation [13].

[1] `magazino.eu` [2] `kbs.informatik.uos.de`
[3] `ruvu.nl` [4] `github.com/magazino/move_base_flex`
[5] `wiki.ros.org/move_base_flex`

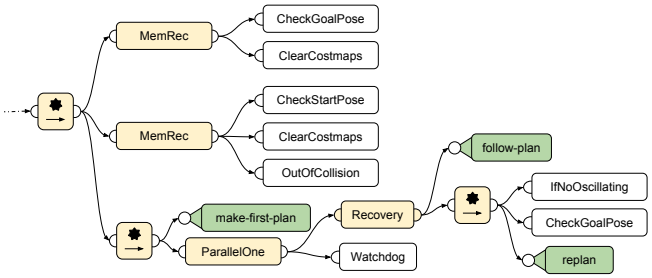

Fig. 5: *Navigation Sub-Tree*: It checks goal and start pose and calls appropriate recovery behaviors if necessary. After that, it performs planning and control using the sub-trees shown Fig. 6 and 7.

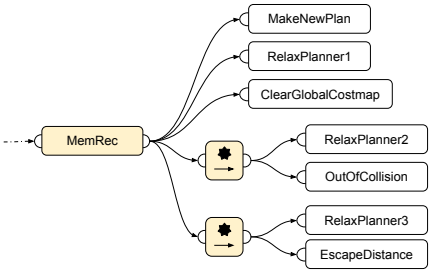

Fig. 6: *Planning Sub-Tree*: It computes an initial path and calls recovery behaviors if necessary.

This can be a starting point for developers to build something more complex then it is possible with `move_base`.

This example stresses the difference in runtime between *move_base* and MBF when they are embedded in bigger scenarios. MBF can reuse already computed paths, where *move_base* can not do that. Depending on how complex and detailed the path planning is, it can be important to save that time by avoiding unnecessary replanning.

B. Behavior Tree

Behavior trees (BTs) have shown a notable capacity in modeling complex behavior in several fields in AI and robotics, as described in detail in [10]. They are known for scaling well owing to their hierarchical nature, where simpler operations encoded as sub-trees are composed to create increasingly more complex, higher level behaviors. Next, we briefly explain the node types we have used to compose the navigation BT.

The Leaf Nodes: Leaf nodes (colored in white in Fig. 5, 6, and 7) are either conditions to be checked or actions to be performed in order to achieve some kind of effect in the environment.

Inner Nodes: The inner nodes connect these actions and conditions into the overall task structure. There are two types of inner nodes:

- 'composites' (in yellow), which call children in a particular pattern (parallel, sequence, etc.)
- 'decorators' (in blue), which modify the behavior of a single child.

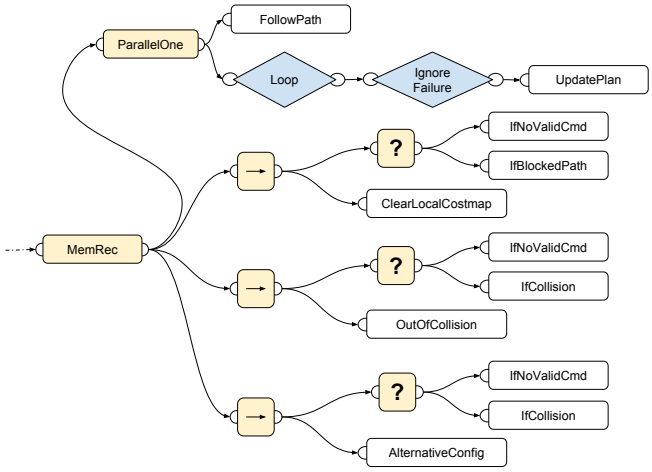

Fig. 7: *Control Sub-Tree*: It executes the controller while updating the plan and calls specific recovery behaviors depending on the controller's *outcome* number code if the controller fails.

Sub-Tree Nodes: The sub-tree nodes (in green) are a special kind of decorator that hierarchically import another tree into the current one.

1) Navigation Sub-Tree: Figure 5 presents an excerpt of the BT used at Magazino. The full tree (55 nodes in total) is too big to represent here. We have omitted the parts dealing with waypoints and data recording for post-mortem analysis.

We start with checking if either the start or goal pose is in collision and, if necessary, try to apply the appropriate recovery behaviors in a sequence: clear the costmaps for both poses, and further escape out of collision if the start pose is blocked. The memory recovery (MemRec) composite ensures we alternate calls to *CheckStartPose* and *CheckGoalPose* with all the pertinent recovery behaviors in sequence, until the check passes, or we exhaust all the behaviors.

2) Planning Sub-Tree: We then call the planning sub-tree, expanded in figure 6, to calculate an initial path. Again, we keep retrying after applying some recovery behaviors: clear global costmap, relax planner constraints, move away from the closest obstacle, etc. Note that we can run combinations of complementary behaviors, if needed.

3) Control Sub-Tree: If we get a valid path, we call with it the follow path sub-tree, expanded in figure 7. It starts the controller in charge of executing the path. Once again, we have controller-specific recovery behaviors that we call in case of failure. Thanks to the extended plugin interface described on III-E, we restrict the applicable behaviors depending on the *outcome* number code reported by the plugin, saving the time expended by the old `move_base` executing useless behaviors and calling again the plugin that, in all likelihood, will fail again.

If the controller keeps failing after trying all the appropriate recovery behaviors, the BT falls back again to the planning sub-tree, but taking care of avoiding oscillation by the alternating calls to planner and controller.

In parallel with the controller, we keep calling a second planner that modifies the initial path as the sensors discover

plugin	*move_base*	MBF
planner	1195	290
controller	389	232
recovery	194	180
clear costmaps	175	93
out of collision	6	46
relax controller	4	31
relax planner	7	10
escape distance	2	0
required time	4926 sec	4312 sec

TABLE I: Plugin calls and the required navigation time.

unmapped obstacles or clear previously seen ones.

In parallel with the path following, we start a watchdog that monitors our progress and detects failures on the motors and whether the robot is oscillating

V. Experimental Results

Though the main contribution of this paper is to introduce MBF and to show its advantages for navigation in ROS, we support our claims with experimental data, as well. The rigidity of `move_base` and conversely the biggest gain in adopting MBF, is best appreciated by the number of useless calls of the planner and recovery behaviors not appropriate for the actual problem. We have accordingly recorded the calls to every plugin loaded by `move_base` and MBF while navigating in a simulated version of a typical warehouse environment.

Our experimental setup consists of a robot traversing 6 waypoints distributed across an environment of size $30 \times 30\,\mathrm{m}$, with 5 moving obstacles (robots of the same model as the one recording data), repeated 10 times.

As shown in Table I, the execution time decreases by 12.5% ($\approx 10\,\mathrm{min}$) from $4926\,\mathrm{s}$ ($\approx 82\,\mathrm{min}$) to $4312\,\mathrm{s}$ ($\approx 72\,\mathrm{min}$). The two main factors accounting to these results are: 1.) The reduction of calls to the global planner (1195 vs. 290). This is particularly relevant in our scenario since we use a relatively slow ($\approx 400\,\mathrm{ms}$ in average) path planner. The reason for this reduction is clear: After many controller failures, we can recover and continue to follow the current path. 2.) The number of different recovery behaviors calls is more homogeneous with MBF. This supports our statement that MBF allows calling the appropriate recovery strategy. By contrast, `move_base` calls disproportionately *clear_costmaps*, the first recovery behavior it tries.

VI. Conclusion & Future Work

We introduced MBF, an advantageous replacement for the `move_base` core component of the ROS navigation stack. Furthermore, we presented the advantages of MBF in comparison to `move_base` [1] and `flexible_navigation` [8]. Besides overcoming limitations, MBF opens ROS navigation to a wide range of possibilities in terms of map representations and navigation strategies. MBF has been successfully used for indoor and outdoor environments with different map representations, such as meshes [12].

We have presented two successful deployments, in the industrial and academic domains. In both scenarios the adoption of MBF has boosted the robots' abilities to navigate very large and highly dynamical environments, noticeably increasing navigation robustness and saving time. After the release, MBF has aroused much interest within the robotics community. We are confident it will become a core component of ROS navigation soon, to the point of replacing the current `move_base`. Moreover, MBF is an ongoing project, and new features are expected in the subsequent releases, among them native support for Ackermann controllers, a pause and resume navigation, more flexible targets, such as target areas, loading and unloading plugins at runtime, concurrently running plugins. Moreover, standard derived MBF servers and interfaces for different map representations are planned, e.g., for `grid_map` [14], OctoMap [15], and navigation meshes [12].

References

[1] E. Marder Eppstein. (2016) move_base - ros wiki. [Online]. Available: http://wiki.ros.org/move_base

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[3] O. S. R. Foundation. (2018) Ros. [Online]. Available: http://ros.org

[4] J. Bohren and S. Cousins, "The smach high-level executive [ros news]," *IEEE Robotics & Automation Magazine*, vol. 17, no. 4, pp. 18–20, 2010.

[5] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered costmaps for context-sensitive navigation," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE, 2014, pp. 709–715.

[6] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," *IEEE transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007.

[7] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, 2010, pp. 300–307.

[8] D. C. Conner and J. Willis, "Flexible navigation: Finite state machine-based integrated navigation and control for ros enabled robots," in *SoutheastCon 2017*, March 2017, pp. 1–8.

[9] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[10] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. ARXIV, August 2017.

[11] P. Schillinger, S. Kohlbrecher, and O. von Stryk, "Human-robot collaborative high-level control with application to rescue robotics," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016, pp. 2796–2802.

[12] S. Pütz, T. Wiemann, J. Sprickerhof, and J. Hertzberg, "3d navigation mesh generation for path planning in uneven terrain," in *9th IFAC Symposium on Intelligent Autonomous Vehicles (IAV 2016)*. IFAC, 2016.

[13] S. Pütz. (2018) uos/ceres_robot: ceres robot ros driver and tools. [Online]. Available: https://github.com/uos/ceres_robot

[14] P. Fankhauser and M. Hutter, "A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation," in *Robot Operating System (ROS) – The Complete Reference (Volume 1)*, A. Koubaa, Ed. Springer, 2016, ch. 5. [Online]. Available: http://www.springer.com/de/book/9783319260525

[15] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013, software available at http://octomap.github.com. [Online]. Available: http://octomap.github.com