Better Lost in Transition Than Lost in Space: SLAM State Machine

This paper has been published on the IEEE/RSJ International Conference on Intelligent Robots and Systems.
Please cite this work as:

```
@inproceedings{colosi2019better,
  title={Better Lost in Transition Than Lost in Space: SLAM State Machine},
  author={Colosi, Mirco and Haug, Sebastian and Biber,
          Peter and Arras, Kai O and Grisetti, Giorgio},
  booktitle={2019 IEEE/RSJ International Conference on
             Intelligent Robots and Systems (IROS)},
  pages={362--369},
  year={2019},
  organization={IEEE}
}
```

Better Lost in Transition Than Lost in Space: SLAM State Machine

Mirco Colosi Sebastian Haug Peter Biber Kai O. Arras Giorgio Grisetti

***Abstract*— A Simultaneous Localization and Mapping (SLAM) system is a complex program consisting of several interconnected components with different functionalities such as optimization, tracking or loop detection. Whereas the literature addresses in detail how enhancing the algorithmic aspects of the individual components improves SLAM performance, the modal aspects, such as when to localize, relocalize or close a loop, are usually left aside. In this paper, we address the modal aspects of a SLAM system and show that the design of the modal controller has a strong impact on SLAM performance in particular in terms of robustness against unforeseen events such as sensor failures, perceptual aliasing or kidnapping. We preset a novel taxonomy for the components of a modern SLAM system, investigate their interplay and propose a highly modular architecture of a generic SLAM system using the Unified Modeling Language™ (UML) state machine formalism. The result, called SLAM state machine, is compared to the modal controller of several state-of-the-art SLAM systems and evaluated in two experiments. We demonstrate that our state machine handles unforeseen events much more robustly than the state-of-the-art systems.**

I. Introduction

SLAM is a fundamental skill for mobile robots in various application domains. After decades of research and significant progress, SLAM has become a mature and well understood problem with a commonly used system decomposition into front- and back-end and problem formulation as a graph optimization and probabilistic estimation task, see e.g. Cadena *et al.* [1]. However, despite these advances, most state-of-the-art SLAM systems are unable to readily deal with the variety of situations that a real robot encounters when exploring an unknown environment autonomously. Unforeseen events such as collision with obstacles, sensor failures, kidnapping, perceptual aliasing or semi-static map changes may occur at any time. Past research has typically focused on the component level to deal with such events, for example by increasing robustness of back-end solvers or data association techniques.

In contrast, we address this problem on a system level as it appears natural to represent and reason about above events as modal states in a system architecture. We tackle the questions of *(i)* how a SLAM system should be designed on architectural and decision-theoretical level and *(ii)* how design choices in that architecture may impact SLAM performance. We analyze prominent state-of-the-art SLAM systems and propose a taxonomy of the components of these systems. In addition to the component view, we highlight for each system the modal aspects that control the interplay between these components. This interplay can be seen as a set of rules that depend on the status of the components over time. We model these rules as an extended finite state machine [2] whose states represent specific modes of a SLAM system such as *localized*, *lost*, or *initializing*. Transitions occur as a response to asynchronous events or verified conditions. Asynchronous events may be raised by user interactions. Conditions are evaluated based on the outcome of the processing modules.

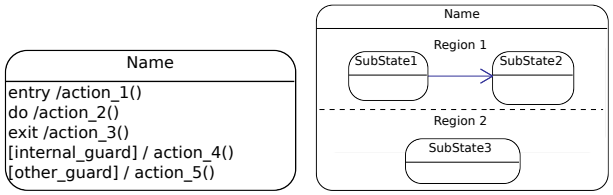
We restrict our study to graph-based SLAM systems and focus on a set of well-known approaches [3], [4], [5], [6], [7], [8], [9]. For each of those systems, we conduct, when possible, both an open- and closed-box analysis. The open-box analysis consists of a literature review and the inspection of the open source implementation, if available. The closed-box analysis is a performance evaluation on multi-sensor datasets that contain relevant levels of difficulties, including rare corner cases. Finally, by combining the strengths while avoiding the weaknesses of related work's state machines, we propose our SLAM state machine as a modal controller for a generic SLAM system and hypothesize that this architecture leads to improved SLAM robustness under wider ranges of experimental conditions. To the best of our knowledge, the only work in the SLAM-related literature which uses a state machine as a behavioral controller is Torres-González *et al.* [10]. In their work, a simple state machine controls the system to reduce resource consumption when the map is already built and the system is affected by low noise level.

II. UML State Machine Notation

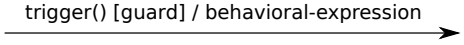
In this section we describe the formalism we used to capture the modal behavior of a SLAM system. As mentioned in the introduction, we rely on an extended finite state machine. Specifically, we formalize the taxonomy and representation of our architecture as a behavioral UML state machine [11]. This formalism supports features such as hierarchically nested states, orthogonal regions, entry and exit actions of states and internal transitions. Using these features leads to a compact and clear representation of the system. We summarize the UML state machine formalism focusing on the concepts used in our work.

The evolution of the system results in a traversal of the states. The system can leave a state and enter a neighboring one according to the traversed transition. The base element of the state machine is the *vertex*, which can be either a *pseudostate* or a *behavioral state*. Pseudostates have predefined behavior encoded in the language. Examples include the *inital* and the *terminate pseudostate* which model the entry and exit points of our state machine. Behavioral states, in contrast, are specified by the user and allow to describe

M. Colosi and G. Grisetti are with the University *La Sapienza* of Rome, Italy. {colosi, grisetti}@diag.uniroma1.it. M. Colosi, S. Haug, P. Biber, K.O. Arras are with Robert Bosch Corporate Research, Stuttgart, Germany. {mirco.colosi, sebastian.haug, peter.biber, kaioliver.arras}@de.bosch.com. This work has been partially supported by Robert Bosch GmbH.



(a) UML Simple State (b) UML Composite State



(c) UML Transition

Fig. 1: Graphical representation of main UML elements

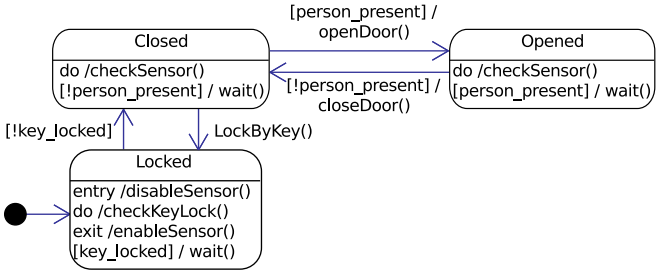


Fig. 2: Simple example of a working automatic door.

an arbitrary set of actions that are executed while in a state. The types of behavioral states we use are:

Simple state: is a vertex without substates. A simple state [Fig. 1a] is defined by a *name*, a set of *internal activities* usually organized in *entry*, *do* and *exit* actions, and a set of *internal transitions*, which are triggered by specific events and protected by boolean guard conditions.

Composite state: is a vertex that contains regions. Each region has a set of vertices and transitions. A composite state [Fig. 1b] is defined by a *name*, a set of regions and an optional pair of *internal activities* of the type *entry* and *exit* action. The contents of different regions are executed in parallel. While in a region of a composite state, it is not possible to migrate in another region of the same state.

Directed relations between a source and a target vertex are called *behavioral transitions* [Fig. 1c]. Traversal is triggered by the occurrence of specified events and is executed if the conditional guard of the transition is verified. Upon traversal an optional behavior is executed.

We provide an example of a simple UML state machine for illustration [Fig. 2]. An automatic door has a sensor that can detect a person approaching the doorway and can be locked with a key. The door can check both the condition of the sensor and the locker. A person with the key can lock the closed door. If the door is locked, the person detector is disabled. When locked, the system polls on the locker status. Unlocking the door activates the person detector and brings the system in the *closed* state. In this state, the system polls the person detector. If a person is present, the door opens and remains in the *opened* state until the person is not present. When this occurs, the door is closed again and the system moves to the *closed* state. From this state, the door can be locked again or it opens upon people arrival.

III. Taxonomy of SLAM Components

A modern graph-based SLAM system consists of several processing modules and a set of shared data structures. In this section, we present a novel taxonomy that characterizes the set of components typical for the SLAM systems considered here.

The primary task of a processing module is to perform computation on the input data and to potentially modify some data structure such as the map. Computations may fail due to defective inputs or processing errors. Each module reports a set of variables that contain the status and the result of the computation to a modal controller (*e.g.* a state machine) which determines the best next action to take.

The map in these systems is represented as a pose graph, where each node represents a geometric transformation. Typically, each node is connected to a local representation of the environment consisting of scans, aggregated scans, point clouds, keyframes or combination of them. Edges in the pose graph denote relative transformation estimates between nearby local maps. These transforms are inferred either directly from the sensor data or by registering local maps between spatially close nodes. We will use the term *local map* to refer to the map information stored in a graph node. Local maps can also contain (or consist of) discrete entities in the world such as geometric primitives or objects to which we refer as *landmarks*. Landmarks can be shared among different local maps.

With reference to Tab. I, the following processing modules are typical in a modern graph-based SLAM system:

Raw Data Preprocessor: is the module that processes raw sensor data and takes care of the synchronization of multiple sensor streams. These operations include odometry interpolation or IMU pre-integration. The raw data preprocessor is also in charge of performing per-datum operations such as feature extraction or denoising. The main purpose of the raw data preprocessor is to present the subsequent modules with data at an adequate level of abstraction, that are spatially coherent and temporally synchronized. We will refer to the output of this component as a *measurement*.

Initializer: operates on the output of the raw data preprocessor by aggregating data until sufficient conditions to start the rest of the system are met. A typical instance of this module in monocular SLAM systems is Structure-from-Motion (SfM) for estimating 3D structure from a sequence of images. The initializer notifies when it is ready by setting the `initialized` flag to true.

Aligner: is in charge of determining the relative transformation between two measurements or between a measurement and a local map. For instance, a monocular system might rely on a point-to-point aligner that registers a set of 3D points onto their image projections, or an ICP algorithm that determines the similarity transform between nearby point clouds. The aligner is the core algorithm that performs the registration and has no notion of state. In a SLAM system multiple aligners might co-exist and are used within more complex modules such as the Tracker or the Loop Detector.

Component	Input	Output	Status/Error Conditions
Raw Data Preprocessor	Raw Measurements	Measurements, Features	-
Initializer	Measurements, Features	Initial Local Map	`initialized` is set on success
Aligner	Local map, Measurements / Local map, Local map	Transform, Statistics	-
Map Merger	Local map, Measurements, Transform	Augmented Local Map	-
Tracker	Local map, Measurements	Transform in Local Map	`able_to_track` set on success
Loop Detector	Global Map, Local Map / Global Map, Meas.	Set of Closure Candidates (spatial constraints)	-
Loop Validator	Set of Closure Candidates	Set of Valid Candidates	`able_to_localize` set to true if a valid loop closure is found
Global Optimizer	Set of Valid Candidates Global Map	Optimized Map	`consistent_map` set false if uneven distribution of resitual error in map or outlier rejection routine fails

TABLE I: Taxonomy of the components of a SLAM system. The table summarizes for each comonent, the input and the output as well as the error/status conditions that are captured by boolean variables affected by the computation.

Map Merger: is the module responsible for integrating the current measurement into the local map upon successful registration. The map merger may carry out straightforward averaging of points, or implement more sophisticated schemes ranging from structure-only Bundle Adjustment (BA) to full BA at local map level. The map merger extends the local map by refining the geometry of the landmarks and augmenting their appearance information.

Tracker: estimates the current position of the robot in the local map. Common visual odometry or incremental scan matching routines fall into this category. They usually rely on an aligner that registers a measurement onto the current local map. In addition to an aligner, a tracker has the notion of pose inside the local map, and can operate multiple sensor modalities. For example, a tracker might perform an update based on odometry when no other measurement is available. The tracker might fail due to an unsuccessful alignment. The status of the tracker is reported by the flag `able_to_track`.

Loop Detector: determines which of the already seen local maps are similar to the current one. Appearance-based loop detectors such as Bag-of-Words (BoW) [12], Hamming Binary Search Tree (HBST) [13] or Fast Laser Interest Region Transform (FLIRT) [14] find sets of feasible local maps whose landmarks' and current measurement appearances match. A further geometric validation is required to reject false matches and retrieve the relative transform between the current measurement and each matching local map. This stage is usually performed by using an instance of aligner, typically with different settings from the aligner used in the tracker. Each loop closure generates a new candidate edge in the graph that is not yet inserted into it.

Loop Validator: is the module in charge of rejecting or accepting loop closure candidates found by the detector. Since a non-valid loop closure can compromise the entire mapping process, the loop validator usually performs a delayed decision, by checking the consensus of a pool of closures that it maintains over time. Typical schemes for the loop closure include RANSAC based approaches [15], or spectral clustering [16]. Using a loop validator is not required for a basic SLAM system, but its presence greatly enhances the system's robustness especially in presence of perceptual aliasing. The loop detector and the loop validator report a successful computation by setting the flag `able_to_localize`.

Global Optimizer: computes the configuration of nodes in the graph that best satisfies the constraints expressed by the edges. The valid closures passed to this module are inserted in the graph. The optimizer can be used also to refine the position of the map's landmarks, as in the case of BA. Nowadays the most common approaches rely on sparse Iterative Least-Squares [17], [18], [19]. An uneven distribution of the residual errors in the graph after optimization is usually a symptom of a wrong loop closure. This is signaled by setting the condition flag `consistent_map` to false. Outlier rejection at graph level such as switchable constraints [20] or Dynamic Covariance Scaling [21] are available in modern optimizers.

We note that not all modules are required for every SLAM systems depending on the type of sensor, the operating conditions or the SLAM approach. However, for some systems, a distinction between module functionalities is not always easily identifiable. If local map construction is done with a Kalman filter, for instance, tracking and map merging are identical as map and robot states are always updated simultaneously with this approach. Similarly, some laser scanner-based systems may not require a raw data preprocessor or an initializer since the tracker operates on raw range data that requires no initialization. If measurements have no appearance information, the loop detector can just exploit graph topology and the uncertainty in the map nodes to select feasible closure candidates. Yet, to the best of our knowledge, this taxonomy generalizes most of the SLAM systems of the current state of the art. Instantiating these components alone, however, is not sufficient as argued in Sec. I, which is why we consider the modal aspects of a SLAM system hereafter.

IV. Analysis of the State of the Art

In this section we analyze prominent SLAM systems with respect to their modal behavior. For each system, we extract the state machine from the respective publication and the open-source implementation, if available. However, not all systems are open-source, and even for those that are, it is not always straightforward to extract the modal behavior just by code inspection. Therefore, we have conducted an additional closed-box analysis of each system to confirm correctness of the reconstructed state machines. For completeness, we put the diagrams extracted from the original papers that informally illustrate the modal system aspects side-by-side with the formal state machine - for clearness's sake, we discard the pseudostates in depicting these state machines. To this end, we ran each system with a standard dataset to which we introduced artifacts to create relevant events and corner cases: we simulate kidnapping by suppressing large portions of the data in the middle of the mission and sensor failures by injecting isolated false measurements at random intervals. We have also chosen a dataset which includes regions with high perceptual aliasing to simulate wrong loop closures and tracking failures.

To report the results of our analysis we consider camera- and laser range-based SLAM systems separately. For the analysis of camera-based systems we use the following datasets: TUM [22] and 2D Cartographer Backpack - Deutsches Museum [23].

A. Camera-Based Systems

We analyzed approaches that operate on monocular and RGB-D cameras, namely: ORB-SLAM2 [3], LDSO [4], ProSLAM [5]. Furthermore we also present the open box analyses of SLAM++ [6] and Fusion++ [7] but we could not perform the closed box analysis due to the unavailability of an open source implementation. We used the same dataset to feed both RGB-D and monocular systems. We handled the monocular case by suppressing the depth channel from the images.

ORB-SLAM2 [Fig. 3] supports both monocular, stereo and RGB-D sensor streams. The first processing step consists in extracting features from the current image. Subsequently, only in the monocular case, an initialization step aiming at constructing the first set of 3D points to perform visual tracking is executed. This step is not required for stereo or RGB-D data. Once initialized, the keypoints and the features are used to track the position of the robot in the current local map. Local maps are organized in a graph, and each time the robot leaves the current local map, a local refinement step is performed on the map landmarks. Notably upon relocalization, local maps can be reentered and refined again. Loop closing occurs by determining candidate closures through BoW [12], and performing a geometric validation. Upon a successful loop closure, an edge is added to the graph which is consequently optimized. In this case the tracker is loaded with the local map resulting from the closure, and the tracking continues from that point. If the tracking fails, the system enters a lost state where it attempts to relocalize in the existing map. The measurements acquired in this interval of time are dropped. Upon user request, the system can disable the map update, resulting in a visual localization engine.

LDSO [Fig. 4] is built on top of the successful DSO system [24] for visual odometry. During initialization, as in the ORB-SLAM2 mono-camera case, it provides a coarse estimation of the pose by SfM computation. When initialized, the mapping phase based on semi-dense visual odometry begins. DSO computes the ego-motion estimation and provides an estimated depth map of high gradient regions. SLAM is achieved by selectively extracting salient features used to annotate the pixels of the keypoints. Loop closure is achieved by using BoW on the extracted keypoints, in a manner as in ORB-SLAM2. Upon loop closure, a global map optimization is triggered. LDSO terminates either upon user request or when the visual odometry fails. On failure no recovery is attempted.

ProSLAM [Fig. 5] operates on both stereo and RGB-D data. Initialization occurs when the system is able to compute the depth from a pair of images. In the RGB-D case, this is verified whenever there are enough features and they are in the depth range of the camera. The tracking is done on local maps, by minimizing the distance between the perceived keypoints and the corresponding ones in a local map. Loop closure is done by using a HBST [13], and reported closures are geometrically validated similar to ORB-SLAM2. Similarly, a global optimization is triggered whenever a valid loop closure is detected. When tracking fails, the system starts creating a new portion of the global map, which is disconnected from the previous one. A successful loop closing might result in rejoining disjoint portions of the global map.

We also studied the SLAM++ [6] and Fusion++ [7] approaches purely based on the literature because no open-source code is available. However, we were able to understand the behavior of the system thanks to the videos and the schematics presented in the latter paper. When the system is able to track, the mapping phase begins, and while it moves in a known location, the system localizes itself. If it gets lost, the system creates a new map with the incoming data and attempts relocalization in the previous map. This results in a behavior similar to ProSLAM.

B. Laser-Based Systems

In this section we focus our analysis on two state-of-the-art systems: Cartographer [8] and SRRG mapper 2D [9]. The first handles multi-echo laser scans while the second operates on simple laser scans. To use the same multi-echo dataset with both systems, we pre-processed the data to provide only the most informative echo of the laser for SRRG mapper 2D. Since no initialization is needed when working with laser scans, both the systems do not have an initialization phase.

The Cartographer [Fig. 6] package offers distinct configurations for localization and mapping tasks. When mapping, the processed data are used to build local maps. The local map update is done via scan matching and the pose of the

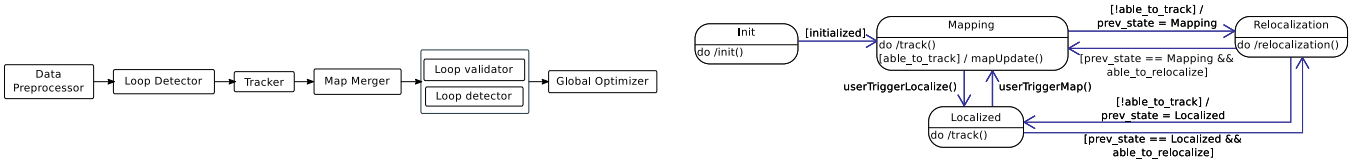


Fig. 3: ORB-SLAM2: On the left the schematic system overview, on the right the reconstructed state machine.

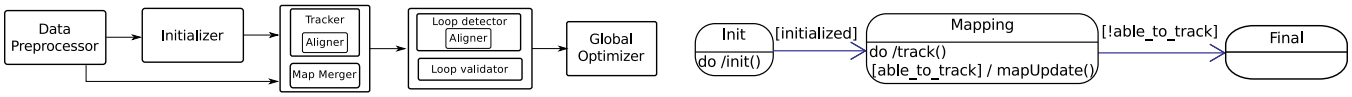


Fig. 4: LDSO: On the left the schematic system overview, on the right the reconstructed state machine.

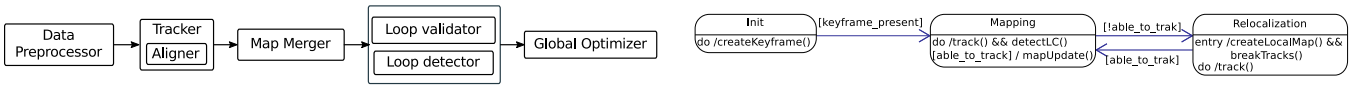


Fig. 5: ProSLAM: On the left the schematic system overview, on the right the reconstructed state machine.

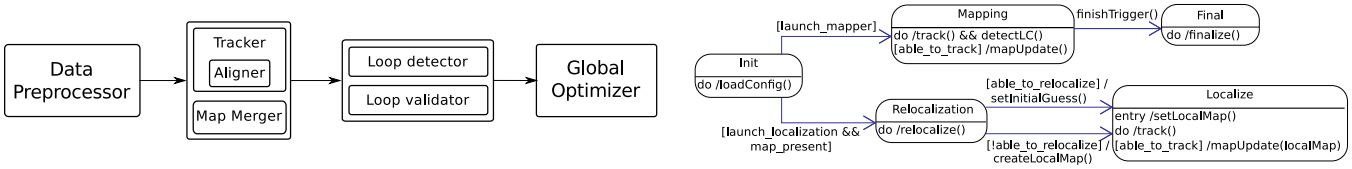


Fig. 6: Google Cartographer: On the left the schematic system overview, on the right the reconstructed state machine.

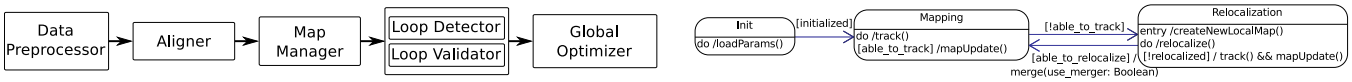


Fig. 7: SRRG mapper 2D: On the left the schematic system overview, on the right the reconstructed state machine.

robot is tracked in the local map. Each local map is stored in the node of a pose-graph. A local map is considered complete when the robot is unable to introduce innovation to the current local map. New nodes are added to the graph upon creation. A loop detection routine searches for candidate closures around the current local map among the neighboring nodes. Upon successfully registration of two local maps, a match is found and the system performs graph optimization. The user can establish when to stop the mapping phase and a final optimization is performed to both the map and the trajectory. If Cartographer is in localization mode, the system attempts to globally localize in the loaded map and sets the robot pose to the initial estimate, if successfully relocalized. Otherwise, the initial guess is set to the origin of the map. From now on, a "relaxed mapping session" is performed, where the system operates on the existing map in tracking mode, without performing side-effect on it.

The SRRG mapper 2D architecture [Fig. 7] relies on local maps consisting of points with normals. These normals are extracted from the raw scan in a preprocessing stage. The logic is similar to Cartographer in mapping mode. In addition to it, this system dynamically prunes overlapping local maps, to limit the dimension of the map.

C. General Considerations

From our analysis, we can argue that most of the SLAM research systems are built to offer the best compromise between processing time, overall precision/accuracy, and performance, bracketing the robustness aspect. Robustness can be achieved when some corner cases are taken into account. Here are the highlights of this study and that we take into account in our following formalization.

Nearly all tested systems suffer when put in corner cases. ORB-SLAM2 and LDSO achieve impressive results in map construction and trajectory accuracy, but they do not fully address a recovery when lost. ORB-SLAM2 always tries to relocalize in the previous created map while neglecting the current measurements. Instead, LDSO does not even attempt relocalization and stops the execution of the program. A possible solution could be the one we find in SLAM++ and SRRG mapper 2D, where a new map is created when the system is lost and then the system tries to merge the current and the previous map to create a most consistent global map.

The topic of localization is more complex than it seems. On the one hand, localizing whenever possible is computationally more efficient. On the other hand, by doing so the system cannot recover from failures since the measurements gathered during localization are dropped. Some approaches such as ORB-SLAM2 and Cartographer let the user choose in which mode to run, but do not automatically switch between the modes during operation. A feasible and safe solution could be to have a localization module running when the system is certain to move in known locations.

Finally, considering that most SLAM systems are executed to obtain a map, having a batch optimization step that refines the estimate to achieve the maximum accuracy is a good feature. Notably, this is done by both Cartographer and ORB-SLAM2.

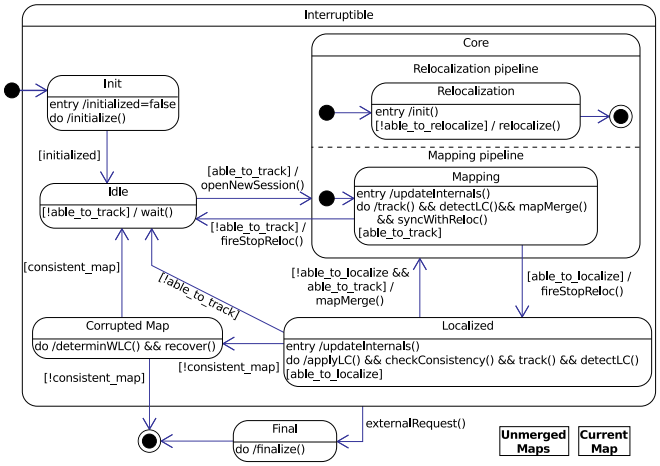


Fig. 8: Our SLAM State Machine.

V. SLAM State Machine

In this section we complete the description of our SLAM architecture [Fig. 8] by describing the state machine used to govern the interplay of the components described in Sec. III. The transitions are governed by the status variables of the components. The states in our architecture are the following:

Interruptible: is the macro-state in which the system lives. The only purpose of this state is to handle the interruption requests triggered by the user or at the end of a mapping session.

Init: captures the case when the system has been just started or reset. In this state the initializer is invoked. The state machine stays in this state until the initialization succeeds (`initialized`). Optionally, the init can preload the SLAM components with map and data from a previous session to continue a suspended or aborted mission.

Idle: the system enters this state when it is `initialized` or a fault occurs - e.g. not `able_to_track` or the `consistent_map` flag was previously false. When the robot is `able_to_track` a new mapping session starts.

Core: is entered as soon as the robot can track, thus the system can both create a map and relocalize itself in the unmerged maps, if any. The mapping and relocalization modules run independently in two orthogonal regions.

Relocalize: is entered when a new session begins. In this state, global localization against already seen maps is continuously attempted. This state can be implemented by combining a loop detector and a loop validator. Upon success the `able_to_relocalize` flag is toggled.

Mapping: is the state in which the robot is exploring an unknown area. In this state, the tracker is called for each new measurement. Upon a successful tracking, it sets `able_to_track` flag and the current local map is augmented by calling the map merger. Regularly, a new local map is generated, added to the actual global map and the tracker reset. Whenever a new local map is generated, a loop closure is attempted by using a loop detector and a loop validator. `able_to_localize` is set to true if any valid closure is found. Moreover, while in this state and a relocalization event occurs, it merges localization and relocalization closures setting the `able_to_localize` flag to true.

Localized: while in this state the robot already knows the surrounding environment. This state is entered when the loop validator reports a valid pool of closures. Once in localized state, all closures are added to the graph, and the global optimizer is called. This results in highlighting potential false closures through graph consistency check. When entering the localized state, the tracker is loaded with the best matching local map resulting from the closures. In this way, the number of nodes is limited by the area explored and not by the length of the trajectory.

Corrupted Map: is entered when an inconsistency in the graph is detected. This may occur when the local map in which the system is tracking does not correspond to the measurements and is a sign that a wrong loop closure occurred in the past. In this state, a routine runs to determine which pool of closures is bad. This process might take time and potentially require to reprocess past data. If the routine fails in identifying invalid loop closures, the map is still inconsistent and the execution of the robot should be stopped. Otherwise, the wrong loop closures are removed and the graph is optimized again. Finally, the idle state is entered.

Final: is entered when the mission is completed, or when the user stops the system. In this state an optional batch map refinement takes place and the result is stored.

VI. Validation

As stated in the introduction, we hypothesize that by addressing the modal aspects of a SLAM system, we can enhance SLAM robustness, i.e., stable or improved performance under wider ranges of conditions. To validate this hypothesis we conduct two experiments in which we evaluate the proposed state machine and compare its behavior and the resulting maps to those of other SLAM systems.

A. Case Study: Robot Vacuum Cleaner

In the first experiment we envision a robot vacuum cleaner that explores an unknown environment and encounters unforeseen events such as sensor failures, perceptual aliasing and kidnapping. Its purpose is to demonstrate that our state machine is able to handle those events correctly and compare its behavior to those of other SLAM systems (Tab. II). We assume the robot's motion is generated by a separate system e.g. an exploration algorithm or motion commands from the user. Steps will be referenced as (N).

Consider Fig. 9, in which a user has bought a robot vacuum cleaner and deploys it at home for the first time. When the robot is turned on, it starts in the *Init* state, loads the configuration file, runs the initialization routine and enters the *Idle* state (0). Then, in (1), it starts exploring the environment towards frontier A in the *Mapping* state. Arrived at A (2), the robot chooses B as the next frontier to explore, realizes that it moves in a known area and enters the *Localized* state. It stays in that state until it reaches B at (3). Then it chooses the corridor as the new frontier, starts

Step	Event	ORB-SLAM2	LDSO	ProSLAM	Cartographer	SRRG Mapper 2D	Our State Machine
0	User turns robot on	**Init**	**Init**	**Init**	**Init**	**Init**	**Init, Idle**
1	Robot explores Room 1	**Mapping**	**Mapping**	**Mapping**	**Mapping**	**Mapping**	**Mapping**
2	Robot drives to corridor	**Mapping**	**Mapping**	**Mapping**	**Mapping**	**Mapping**	**Localized**
3	Robot explores corridor	**Mapping**	**Mapping**	**Mapping**	**Mapping**	**Mapping**	**Mapping**
4	Robot enters dark area	**Relocalization**	N/A	**Relocalization**	N/A	**Relocalization**	**Idle**
5	Robot exits dark area	**Relocalization**	N/A	Mapping	Mapping	Relocalization	**Core (M&R)**
6	Robot keeps on moving	Relocalization	N/A	Mapping	Mapping	Relocalization	**Localized**
7	Robot enters Room 2	Relocalization	N/A	Mapping	Mapping	Mapping	**Corrupted Map**
8	Robot explores Room 2	Relocalization	N/A	Mapping	Mapping	Mapping	**Idle, Core (M&R)**
9	Robot reenters Room 1	Mapping	N/A	Mapping	Mapping	Mapping	**Localized**
10	User kidnaps robot to Room 3	Relocalization	N/A	Relocalization	N/A	Relocalization	**Idle**
11	Robot explores Room 3	Relocalization	N/A	Mapping	Mapping	Mapping	**Core (M&R)**
12	Robot re-enters Room 2	Relocalization	N/A	Mapping	Mapping	Mapping	**Localized**
13	User turns robot off	N/A	N/A	N/A	Final	N/A	**Final**
14	User turns robot on	**Init**	**Init**	**Init**	**Init**	**Init**	**Init, Core(M&R)**

TABLE II: Comparison of SLAM state machine behavior in the case study shown in Fig. 9. States are shown in bold face as long as the behavior is correct or optimal along the course of the experiment. The results demonstrate that the state machines of the alternative systems are unable to handle all events correctly and that this can lead to inconsistent maps as shown in Fig. 10

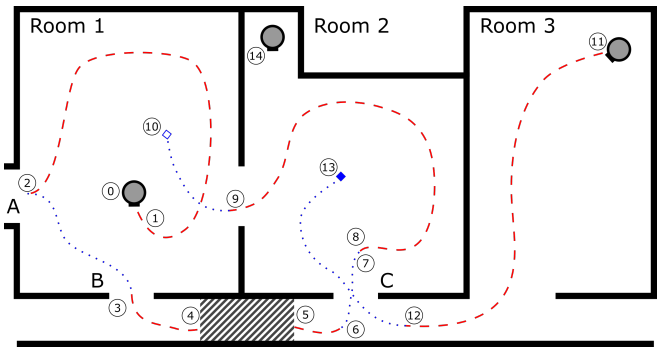


Fig. 9: Case study of a robot vacuum that explores a new environment and encounters unforeseen events such as sensor failures, perceptual aliasing and kidnapping (see explanation in the text and resulting state machine behaviors in Tab. II). Dashed red lines indicate mapping phases, dotted blue lines localization phases.

exploring and enters the *Mapping* state until (4) where the tracker fails, e.g. due to the lack of features or a sensor failure. The robot gets lost, enters the *Idle* state and keeps on moving. At (5) the robot is able to track again and enters the *Core (M&R)* state in which it tries to relocalize in the previously built map and creates a new map in parallel. At (6) the robot is able to relocalize due to perceptual aliasing. It believes to be in the corridor near door B where in fact it is near door C. The state machine enters the *Localized* state and the robot continues by moving towards door C. Entering the room, the robot recognizes a mismatch between its expectation and the current observation (7) which brings the state machine into the *Corrupted Map* state. In this state, the robot starts a recovery routine aiming at detecting and removing the wrong closure. Upon successful recovery, the map is consistent again and the robot enters the *Core (M&R)* state via the *Idle* state at (8). When the robot reaches Room 1 (9), it recognizes the previously explored area, enters the *Localized* state and merges the maps. At (10), the user kidnaps the robot and takes it to Room 3. From one frame to the next, measurements become inconsistent, the tracker fails to localize and the state machine enters the *Idle* state. The robot is able to track again at (11) entering the *Core (M&R)* state in which it simultaneously performs relocalization and mapping until (12) where it relocalizes itself near door C and enters the *Localized* state. When the robot reaches (13), the user decides that the robot worked enough for now and turns it off. The system finalizes and stores the map in the *Final* state. The robot vacuum is placed in the corner in Room 2 at (14). The next time it will be turned on again, after initialization and map loading in the *Init* state, the state machine will transition to the *Idle* and then the *Core (M&R)* state, in which it will successfully relocalize based on the actual sensor readings.

B. Simulated Map Experiment

In the second experiment we demonstrate the impact of an incorrect or suboptimal state machine behavior, as shown in (Tab. II), onto the mapping result. We implement a simple SLAM system controlled by our state machine. In a simulated map shown in Fig. 10, left, we deploy a virtual robot equipped with a 2D laser range finder. Concretely, the components of our SLAM system are as follows:

- **Raw Data Preprocessor:** We compute local approximations of normals to the (x, y)-points, used in the Tracker.
- **Tracker:** For both states, *Mapping* and *Localized*, tracking is implemented as a 2D variant of the NICP algorithm [25].
- **Map Merger:** We implemented the cloud merging procedure in [9], used in the *Mapping* state.
- **Loop Detector:** For the *Mapping*, *Localized* and *Relocalized* states, we implemented an **Aligner** that computes correspondences using K-Nearest Neighbors supported by a geometrical validation test between nodes in the graph.
- **Global Optimizer:** For this component we use g2o [17].
- The *Relocalized* state implements an instance of Monte Carlo localization [26].

The **Initializer** and **Loop Validator** components as well as the *Corrupted Map* and *Final* states were not needed for the purpose of this experiment.

The robot starts at position (0), gets kidnapped at position (1) and taken back within the already mapped area to (0). The second time, the robot gets kidnapped again at position (1) and relocated outside of the mapped area at position (2). The map built by our system is compared

Fig. 10: From left to right: Simulated ground truth map with exploration path, map generated by Cartographer, map generated by SRRG mapper 2D, map generated by our system. Albeit the individual components of our system are rather standard, it can cope with unforeseen events thanks to the proposed SLAM state machine.

to two laser-based SLAM system, namely Cartographer and SRRG mapper 2D [Fig. 10]. Only the map generated with our system is topologically correct. Both Cartographer and SRRG mapper 2D are unable to handle the kidnapping events leading to tracking failures and a corrupted map far away from the ground truth.

VII. Conclusions

In this paper we present a taxonomy for the components of a modern SLAM system and investigate the interplay between them. To this end, we review several state-of-the-art graph-based SLAM systems and highlighted a set of common functionalities and behaviors. We use the UML state machine formalism as an approach to represent and control the interplay of the components and propose a highly modular architecture of a generic SLAM system which we call SLAM state machine. We propose and evaluate an instantiation of this architecture and demonstrate in two experiments that our state machine handles unforeseen events such as sensor failures, perceptual aliasing and kidnapping more robustly than the state-of-the-art SLAM systems.

References

[1] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J.J. Leonard. Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age. *IEEE Trans. on Robotics (TRO)*, 32, 2016.

[2] Kwang-Ting Cheng and Avinash S Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*, 1993.

[3] Raul Mur-Artal and Juan D Tardós. ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. *IEEE Trans. on Robotics (TRO)*, 33, 2017.

[4] Xiang Gao, Rui Wang, Nikolaus Demmel, and Daniel Cremers. LDSO: Direct sparse odometry with loop closure. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*. IEEE, 2018.

[5] Dominik Schlegel, Mirco Colosi, and Giorgio Grisetti. ProSLAM: Graph SLAM from a Programmer's Perspective. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2018.

[6] Renato F Salas-Moreno, Richard A Newcombe, Hauke Strasdat, Paul HJ Kelly, and Andrew J Davison. SLAM++: Simultaneous Localisation and Mapping at the Level of Objects. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2013.

[7] John McCormac, Ronald Clark, Michael Bloesch, Andrew Davison, and Stefan Leutenegger. Fusion++: Volumetric Object-Level SLAM. In *Proc. of the Intl. Conf. on 3D Vision (3DV)*, 2018.

[8] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2D LIDAR SLAM. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2016.

[9] M. T. Lázaro, R. Capobianco, and G. Grisetti. Efficient Long-term Mapping in Dynamic Environments. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2018.

[10] Arturo Torres-González, Jose Ramiro Martinez-de Dios, and Anibal Ollero. An adaptive scheme for robot localization and mapping with dynamically configurable inter-beacon range measurements. *IEEE Sensors Journal*, 14, 2014.

[11] OMG Available Specification. OMG unified modeling language (OMG UML), Superstructure, V2. 1.2. *Object Management Group*, 70, 2007.

[12] Dorian Gálvez-López and J. D. Tardós. Bags of Binary Words for Fast Place Recognition in Image Sequences. *IEEE Trans. on Robotics (TRO)*, 28, 2012.

[13] Dominik Schlegel and Giorgio Grisetti. HBST: A Hamming Distance Embedding Binary Search Tree for Feature-Based Visual Place Recognition. *IEEE Robotics and Automation Letters (RA-L)*, 3, 2018.

[14] Gian Diego Tipaldi and Kai O Arras. FLIRT Interest Regions for 2D Range Data. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2010.

[15] M. T. Lázaro, L. M. Paz, P. Piniés, J. A. Castellanos, and G. Grisetti. Multi-Robot SLAM using Condensed Measurements. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2013.

[16] Edwin Olson, Matthew R Walter, Seth J Teller, and John J Leonard. Single-Cluster Spectral Graph Partitioning for Robotics Applications. In *Proc. of Robotics: Science and Systems (RSS)*, 2005.

[17] Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. g2o: A general framework for graph optimization. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2011.

[18] Michael Kaess, Ananth Ranganathan, and Frank Dellaert. iSAM: Incremental smoothing and mapping. *IEEE Trans. on Robotics (TRO)*, 24, 2008.

[19] Sameer Agarwal, Keir Mierle, and Others. Ceres Solver. `http://ceres-solver.org`.

[20] Niko Sünderhauf and Peter Protzel. Switchable constraints for robust pose graph SLAM. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2012.

[21] P. Agarwal, G.D. Tipaldi, L. Spinello, C. Stachniss, and W. Burgard. Robust Map Optimization using Dynamic Covariance Scaling. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2013.

[22] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A Benchmark for the Evaluation of RGB-D SLAM Systems. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2012.

[23] Google. 2D Cartographer Backpack Deutsches Museum, 2016.

[24] J. Engel, V. Koltun, and D. Cremers. Direct Sparse Odometry. *IEEE Trans. on Pattern Analalysis and Machine Intelligence (TPAMI)*, 40, 2018.

[25] J. Serafin and G. Grisetti. NICP: Dense Normal Based Point Cloud Registration. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2015.

[26] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Monte carlo localization for mobile robots. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 1999.