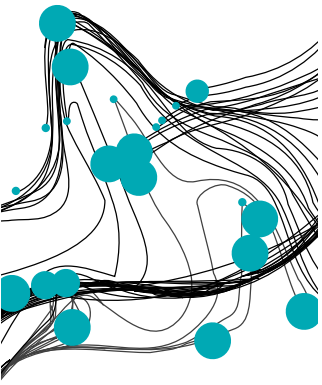
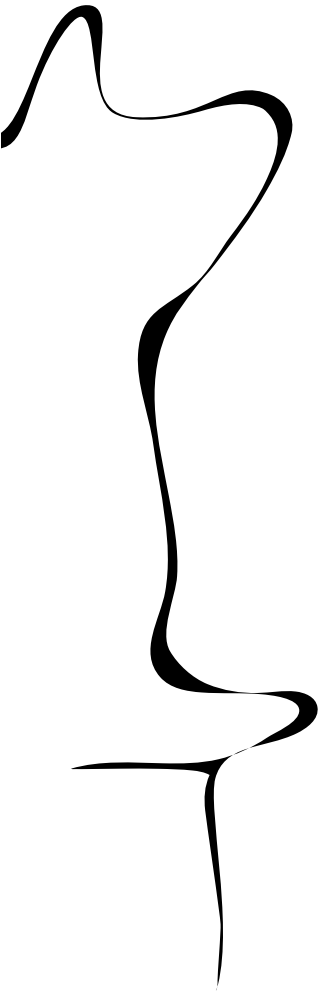


RAM ROBOTICS AND MECHATRONICS



INSIGHTS IN SLAM ALGORITHM MODULARITY FROM RTAB-MAP AND ORB-SLAM2

M. (Mark) te Brake

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
dr. ir. D. Dresscher
dr. F.C. Nex

April, 2021

019RaM2021
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Summary

The reusability and adaptability of SLAM software implementations in general is limited. As a consequence, the use of SLAM systems to support different applications or the adaptation of SLAM systems to different environments or platforms, requires significant efforts.

In order to improve on the reusability of SLAM implementations, this thesis considers two open-source visual, feature-based, pose graph SLAM systems: RTAB-Map and ORB-SLAM2. Several aspects of these systems are investigated to define a baseline for the behaviour and concepts implemented by those SLAM implementations. The analysis uncovers details about the functionality that is contained within these systems, the type of information that is processed for these functions and architectural aspects of both implementations that affect reusability and modularity.

The insights gained by this analysis are used to define a framework of generalised components, which can be used to implement modular SLAM algorithms. Within this framework, a set of three entities (Feature, Sample and Landmark) is used as foundation from which knowledge in a SLAM system is constructed. Correspondences of six different types allow to distinguish all possible relations that can be established between the presented entities.

Six different responsibilities are defined from the analysis of SLAM theory and the systems mentioned before. These responsibilities are assigned to be fulfilled by components of the X-, F-, C-, OE-, DM- and DS-types. The tasks that they perform are in corresponding order: pose estimation for samples and landmarks, feature extraction, establishing correspondence, trajectory and map optimisation and estimation, data management and finally, data storage. Modularity and interchangeability of components is facilitated by the framework by restricting the output of components to a specific format that matches with their responsibilities. This allows for easy interchanging or reuse of components of the same type.

The framework is used to design a modular version of RTAB-Map, as an example on how a real SLAM system can be implemented in a way that promotes the reuse of (parts of) the algorithm.

Contents

1 Introduction

This master thesis is the result of an assignment at the Robotics and Mechatronics group at the University of Twente. The research group takes part in the i-Botics innovation centre, which is a collaboration between the University of Twente and the Dutch TNO institution. As part of the research for inspection robots, i-Botics investigates remotely-operated or tele-robotics.

A key goal of this research is to be able to have the human operator separated at a distance from the robot that he controls, but provide him with an experience that feels as if he is right at the spot. This technology is especially valuable for applications where people have to perform tasks in dangerous environments, or situations where specialists are not locally available to perform difficult tasks.

Remote robotic operations often require some form of map or reconstruction of the environment where the robot is operating in. Of course, it is also important to know where the robot is actually located within this map. This combination of information can be used for navigation, interaction with objects in the environment or for providing the operator with a virtual view into the environment.

If both the map of the environment and the location of the robot have to be estimated at the same time, the task becomes a Simultaneous Localisation and Mapping (SLAM) problem[1]. Although SLAM is a well-researched problem for decades and the theoretical background is extensively established, it is still not trivial to use a SLAM system within a given application.

1.1 Problem statement

The user has to adapt the SLAM system to suit the application and platform at hand, such that the output fits the requirements for further use by the actual application. This involves tuning of parameters, integration of sensor hardware and modifications to the algorithm.

This process is troublesome due to several issues:

- The user does often not have an extended expertise in SLAM systems, but is only interested in using the map or positional information.
- Existing SLAM systems are lacking good documentation. Academic papers describe a concept, method or idea, but do not necessarily provide a comprehensive user manual or software documentation.
- Open source SLAM systems are a large piece of monolithic software, which makes it difficult to isolate the desired functionality or limit the scope of any modifications.

Although some level of understanding will always be required to identify the limiting factors within the system, SLAM systems can improve to facilitate easier reuse and adaptability to new applications.

The goal of this work is to reduce the effort that is required to integrate or adapt an existing SLAM software product to a new application and to facilitate easier exchange of functionality between different SLAM implementations. The solution is sought in a framework of composable components, which should help to isolate and decouple functionality by clearly defined boundaries. These boundaries at the component level also contribute to reusability by confining the required knowledge for understanding the workings of a component and compartmenting the documentation.

[1] Contrary to a pure localisation problem, where the robots location is determined with respect to a given reference. On the other hand, only a mapping or reconstruction problem arises if the robots location can be measured accurately.

1.2 Research questions

While other attempts at solving this problem focused on software engineering or fundamental SLAM theory, this thesis approaches the problem at the system level and looks into the desired functional behaviour. The focus of this work will be on the use of SLAM software by users who like to run or modify existing algorithms, but the solution should also be feasible from the standpoint of the SLAM algorithm developer.

In order to develop a solution which fulfils these goals, the following questions will be addressed in this thesis:

1. What are the functions performed by current state-of-the-art SLAM systems?
2. What kind of information is required or processed in order to perform these functions?
3. In which way does the architecture of these SLAM systems influence the modularity and reusability of the associated functionality?
4. How can functional behaviour be structured in logical framework components and interfaces to facilitate reuse, interchangeability and adaptability?

1.3 Related work

Work relevant to the subject covered by this thesis falls into several categories, which cover reusability efforts for SLAM algorithms, software engineering and frameworks for software in robotic applications. Literature about SLAM algorithms or the theoretical foundations usually do not cover architectural considerations or design patterns. Software frameworks targeted at performing functions for SLAM algorithms exist, such as g^2o (Rainer Kümmerle (2011)) and GTSAM. However, the related papers and documentation are mostly concerned with the workings and usage of those libraries, instead of the architecture of the SLAM system in which it is used.

The work of Ellery (2017) targets the reusability of software for robotic applications in general. He applies three software engineering paradigms (object-oriented programming, component-based software development and the separation of five concerns) to construct a framework for reusable software. Reusability is measured by defining and evaluating reuse readiness levels for 9 different topics: documentation, extensibility, intellectual property, modularity, packaging, portability, standards compliance, support and verification and testing. A component-based, middleware independent implementation of a PID-controller is provided, demonstrating the proposed structure and interfaces. The author also highlights the importance of well-defined interfaces, documentation and manuals.

Abdelhady (2017) and Minnema (2020) develop a component-based framework by identifying commonalities in functionality and interfaces among SLAM algorithms. Both make the distinction between idiothetic and allothetic information. Abdelhady constructs a software product line, by identifying fixed and variable components within SLAM algorithms. By defining these stand-alone components, the number of dependencies through the algorithm is reduced. Proof-of-concept implementations of basic extended Kalman filter and particle filter SLAM are demonstrated, as well as two alternative structures for components to be invoked. Minnema's work targets the interchangeability of sensor and SLAM back-ends. He defines three types of sensors, a generalised structure for a SLAM system and investigates the information that is exchanged at the boundaries between the different parts. This knowledge is used to derive interfaces between the components in the system. A distinction is also made between an idiothetic update step, for processing of incoming idiothetic data and an allothetic update step. This separation works well for the proof-of-concept systems that are constructed, but

may cause difficulties for more complex algorithms that may contain additional dependencies between idiothetic and allothetic information.

Broenink (2016) also considers the vast amount of effort required to adapt and deploy SLAM algorithms to platforms or environments that are different to those used for the original application. The belief or estimation about the real world is considered a common factor through the whole SLAM system. In order to facilitate easier change of sensors, robotic platform or environment, the components of a SLAM system are separated into five functional blocks with clearly defined interfaces:

- Robot specific.
- Environment specific.
- Feature specific.
- Sensor specific.
- World specific.

The proposed structure is used to implement a 2D particle filter SLAM system with wheel odometry and a lidar sensor on top of the ROS middleware. Although the system was able to provide a reasonable map of the environment, further work is required with respect to the map component interface, placement of the feature detection within the structure and issues with the robustness of the feature detection. Furthermore, the system also lacked computational performance optimisations and its contribution to reusability was not validated.

The RobMoSys project (RobMoSys (2017)) aims to develop an open ecosystem for the development of industry-grade software for robotics. It considers a wide range of systems and software engineering aspects. Within the ecosystem three tiers are defined, each with their own roles, requirements and level of abstraction. The problem covered by this thesis relates to tier 2 and tier 3: domain experts define service definitions for SLAM, providing a structure suitable to the SLAM problem. These definitions apply to data structure, properties and communication semantics and facilitate in separated services which allow for more flexibility through composition.

Tier 3 users are then able to apply these service definitions to implement their components. Composition is a key element within the ecosystem to promote reusability. The project also highlights the importance of separation of levels and separation of concerns, and the need for architectural patterns. Although all of these aspects are naturally relevant to this thesis, a closer look at the defined levels is of particular interest:

- Mission.
- Task plot.
- Skill.
- Service.
- Function.
- Execution container.
- Operating system/middleware.
- Hardware.

From the top down, the mission and task plot levels are giving rise to the need for a SLAM skill and are (by themselves) out of the scope of this thesis. As this thesis attempts to stay away from hardware, platform and implementation specific details, the skill, service, function and execution container levels are most relevant for consideration.

1.4 Modularity and reuse for software engineering

From the fields of software engineering and computer science, the subject of reusability is vastly covered. Reusability is also a reoccurring theme in discussions about software for robotic applications. The work of Ellery (2017) (and Anguswamy (2013) to which he refers) touches on several topics that are relevant to writing reusable software. Similarly, books like Cooling (2003) also treat reusability in the context of software development and at a much wider level from a systems engineering point of view.

This Section considers a number of topics that are often encountered when discussing the reusability of software. Designing and writing reusable software involves finding a suitable compromise between these topics that fits the application at hand. These topics are sometimes closely related to each other, which means that they tend to share common goals or values. Although they are listed as separate topics, the boundary is often much more fluid in practice.

1.4.1 Hierarchy and modularisation

Dividing a software application into separate modules aids by reducing the size of a task or problem, such that each module can cope with a specific aspect of the application. By applying a hierarchical structure to organise modules at different levels or layers, the role and level of responsibility of specific modules within the system can be indicated. These concepts help to cut a larger problem into smaller problems and assign responsibilities in a logical fashion.

When discussing modularisation, the words "module", "component" and "object" are often encountered. These words do already impose some form of hierarchy, as they describe entities ranging from a high to low level respectively.

Modularisation of software improves reusability, by defining the boundaries around a certain task or functionality. This allows for existing modules to be taken and used in other systems, which is the key goal for the framework presented in this thesis.

For modularisation to be successfully contributing to reusability, it needs to go hand in hand with the concepts of abstraction and interfaces, which will be discussed next.

1.4.2 Abstraction

The concept of abstraction is to only show the details that matter and hide anything that is not relevant. Of course, the difficult part of abstraction is to decide, given the circumstances, what matters and what not. Abstraction is about considering which information is relevant to carry out the responsibilities that are assigned to a certain hierarchical level or a specific module, component or object.

Reusability is improved by abstraction, as the irrelevant details are hidden from both the programmer as well as the software that is surrounding the functionality that is targeted for reuse. The relevant details that are left, provides a boundary for reuse with reduced complexity.

1.4.3 Interfaces

Interfaces represent the inputs and outputs through which information flows in and out of entities, like the modules, components and objects mentioned earlier. A formal description of an interface specifies in which way information is allowed to cross the boundary of such an entity. Abstraction and interfaces are closely related, as the interface provides a mechanism to

present the details that are relevant to the outside world, while hiding irrelevant concepts and details.

An important property of interfaces is restrictiveness. This means that an interface only carries the bare minimum of information that is required. Restricting interfaces carefully has several benefits:

- Entities cannot access data that they were not intended to.
- Rights to read and write information can be enforced, such that data is only modified by components that are intended to do so.
- Restricting interfaces to carry only the required information prevents unnecessary dependencies when reusing a piece of software.

Reusability is improved by specifying interfaces at the boundary of a module, component or object, such that the programmer knows how to interact with it and get it to perform the operation it should do. A big issue for modularity and reuse arises when interfaces cause so-called leaky abstractions. This happens when interfaces carry information that traces back to implementation specific or low-level details that needed to be abstracted away. These leaky abstractions often result in additional dependencies between different entities due to knowledge or functionality that is now required to be existing at both ends of the interface.

1.4.4 Composition and variability mechanisms

Composition is the process of connecting separate modules or components to construct a larger system. This relies heavily on the previously mentioned topics, as composition requires entities with well defined boundaries and interfaces.

Composition enables a system which supports different configurations, operating modes, platforms or sensors. Variability mechanisms specify and control those options and the required configuration for a given use case.

In this thesis, variants are encountered at the level of the components of which a SLAM algorithm is constructed, the choice of sensory input that is supported, the platform on which the SLAM algorithm is executed and the state it is running in. Reusability benefits from well defined variability mechanisms, such that it it clear which options are available and how and when different variants can be chosen. This involves consideration of several design choices, such as:

- Do two variants of a similar operation result in two different components, or can they be implemented in one component that can be configured as one of both variants.
- Are variants selected and configured at compile-time or run-time?
- Are variants statically configured, or can variants be changed dynamically while the program operates?

1.4.5 Documentation

In order to be able to reuse any piece of software, it should be clear in the first place what that piece actually is, what it does and how it should be used. These questions should be answered by documentation accompanying the software. Documentation can come in several forms, such as a separate document or web page, comments that are added to the code itself and in the form of self-documenting code.

Although this thesis does not treat the subject of documentation in more detail, any of the previously mentioned software engineering concepts contributes to writing documentation in

a meaningful way. The development of a modular framework in this work and the associated language to talk about the concepts that contribute to this framework, provides tools to write about implemented functionality and facilitates consistent and structured documentation.

1.5 Thesis outline

Chapter 2 first introduces some relevant concepts from SLAM theory and then investigates the functionality that is present in a selection of two SLAM systems (RTAB-Map and ORB-SLAM2). This provides a baseline for behaviour that should fit within the developed framework, answering the first two research questions. The third research question is answered in the last Section of this Chapter, which discusses the architectural aspects of the analysed systems with respect to the modularity and reusability of those systems.

In Chapter 3 this information is combined to derive a logical separation of functional behaviour into generalised framework components. These components are accompanied by a specification for interfaces to connect them together. The Chapter also contains guidelines which can be followed to assign functionality to an appropriate component. A demonstration on how to use the proposed framework is given by two examples, one of which covers the RTAB-Map algorithm as analysed in Chapter 2.

In Chapter 4, the conclusions to the research questions and this work are presented, followed by an discussion of the implications and limitations of the presented framework and suggestions for future work.

2 Inside SLAM

This Chapter investigates the functional behaviour and dependencies within existing SLAM systems. Section 2.1 first explains SLAM concepts that are relevant to the systems investigated in this thesis. Section 2.2 then analyses current open source SLAM implementations in order to establish a baseline of functional behaviour that is required by these systems.

2.1 Theory

SLAM is subject of academic research since the 1980s. The first two decades of research are identified as "the classical age" by Cesar Cadena (2016). During this period, the probabilistic formulation of the SLAM problem was developed and the theoretical groundwork for localisation and mapping was laid down. SLAM approaches using (extended) Kalman filters or particle filter approaches also originated from these times.

The book Probabilistic Robotics (Sebastian Thrun (2005)) provides a convenient overview of the related theory and introduces relevant subjects such as probabilistic filter techniques, state estimation and the localisation problem. This thesis follows the notation as introduced in this book.

SLAM essentially boils down to generating a map with environmental landmarks and estimating the current position within this map. This creates a problem with a circular dependency, because mapping requires knowledge about the current location and localisation requires the existence of a map. For regular mapping or localisation, these inputs are assumed to be known beforehand. In literature, the SLAM problem is usually formulated as a probabilistic problem. This takes the form of a posterior probability density function that has to be solved or approached in order to provide the robots location and a map of the environment.

The SLAM problem is commonly divided in two categories: the online SLAM problem and the full SLAM problem. The difference between these two formulations is in the state x that is estimated. While for online SLAM only the current position is estimated each iteration, full SLAM attempts to provide a consistent estimate over the whole trajectory the robot has travelled.

The online SLAM problem can be formulated as:

$$p(x_t, M | z_{1:t}, u_{1:t})$$

Time is defined at discrete intervals, indicated by the indices at the different variables. This posterior probability density function p at time t provides the current estimate of the robots position x_t and the map M containing the estimated landmark positions. These estimates are determined given the set of all measurements $z_{1:t}$ and all controls $u_{1:t}$. The name "controls" refers to commands that are given to the robot in order for it to move. It should be noted that data about the self-motion of the robot from sensors such as odometers, wheel rotation sensors, inertial measurement units and gyroscopes, is also part of the control u_t.

Similarly, the full SLAM problem provides an estimation for the whole trajectory ($x_{1:t}$) and can be formulated as:

$$p(x_{1:t}, M | z_{1:t}, u_{1:t})$$

These formulations occur in many alternate forms in literature, up to the point that they are reduced to the estimation of a single probabilistic variable given a single set of input data (such as $p(X|Z)$ in Cesar Cadena (2016)). In this thesis, the more explicit variant is used, in order to provide more insight in the data that is used.

The definitions mentioned above provide an intuitive description of the task a SLAM algorithm should fulfil, but they hide many details about the actual algorithm. Algorithms often require a

mechanism to determine correspondences between different measurements, observations or other pieces of information. These operations are generally called data association and can be made explicit by formulating the SLAM problem with unknown correspondences c_i:

$$p(x_t, M, c_t | z_{1:t}, u_{1:t})$$

$$p(x_{1:t}, M, c_{1:t} | z_{1:t}, u_{1:t})$$

But it quickly becomes difficult to provide a convenient problem formulation for more elaborate implementations which depend on additional internal states and rely on previous estimates during later iterations.

A special type of correspondences that is often encountered is loop closure detection, or long-term data association. Detecting loops in the measurements that observe the environment helps to identify errors in the trajectory of the robot which may have been accumulated over time.

Finally, it should be noted that all data sets with indices $1:t$ can grow indefinitely in time. Real-world SLAM approaches should include mechanisms to cope with this growing size of data to maintain scalability in time.

2.1.1 Features and landmarks

The terms features and landmarks are commonly encountered in literature when discussing SLAM systems that use camera sensors and/or construct feature-based maps. Features are extracted from the raw sensor measurement by applying an extractor function f:

$$F_t^{1:N} = f(z_t)$$

Feature-based approaches extract a limited number of features from a measurement with a higher dimensionality. Therefore, the amount of information with respect to the original measurement is reduced in favour of a computational advantage.

The sort of feature that can be extracted from a measurement depends on the sensor type. For image sensors, common features are representing edges and corners, fixed patterns such as fiducial markers or objects of a distinct appearance. Range finding sensors can be used to extract features such as lines, corners or local minima. Features can represent a point or larger objects and shapes. In robotics it is also common to identify features in terms of high-level concepts such as hallways or intersections.

A feature consists of a value identifying its location with respect to the reference frame of the robot or sensor, together with a signature vector which holds values that describe one or more properties of this feature.

Landmarks are objects or distinct features in the physical environment of the robot. Sebastian Thrun (2005) uses the terms "feature" and "landmark" interchangeably, as seems to be common practice in literature. As described before, a feature-based map holds a set of landmarks and their estimated locations in the global coordinate frame of the environment that is mapped. So when does a feature become a landmark in terms of the SLAM algorithm? Without a better definition from a reputable source, this work follows the following definition:

A single feature which is extracted from one or more measurement(-s), becomes a landmark when a pose (estimate) in the global coordinate frame of the map is assigned to this feature.

This definition allows for SLAM systems that use landmarks while generating a non-feature-based map.

2.1.2 Pose graphs

Graph-based SLAM solves the full-SLAM problem by constructing a pose graph of which the nodes represent the robot poses or landmark poses (Grisetti (2010)). The nodes in the graph are connected by edges, which represents a metric constraint between the attached nodes. These graphs can be efficiently optimised to find a set of poses for all nodes which minimises the error with respect to the constraints set by the edges.

An example of a scene which can be represented by a pose graph is shown in Figure 2.1. This particular robot travelled along a trajectory from the left to the right, between two walls.

While moving, the robot position is tracked by some kind of (inaccurate) source of odometry information. These estimated poses are indicated by the squares, which can be carried over as nodes in the pose graph. Edges between these nodes are represented by the dashed lines. The associated constraints describe the displacement, calculated directly from the estimated poses. Of course, at this point the constraints match exactly with the poses and optimisation useless.

This changes if the landmarks and corresponding measurements are added to the pose graph. The robot was able to detect several landmarks and estimate their position, as indicated by the circles. These landmark poses can also be added as nodes to the pose graph. The dotted lines represent the associated edges, of which the constraint is based on the measurement that was used to observe the environment of the robot.

The resulting pose graph now contains information from several sources, with different levels of reliability. A pose graph optimisation algorithm is now able to asses the estimated poses and constraints and generate a new arrangement of poses that fits the full set of constraints.

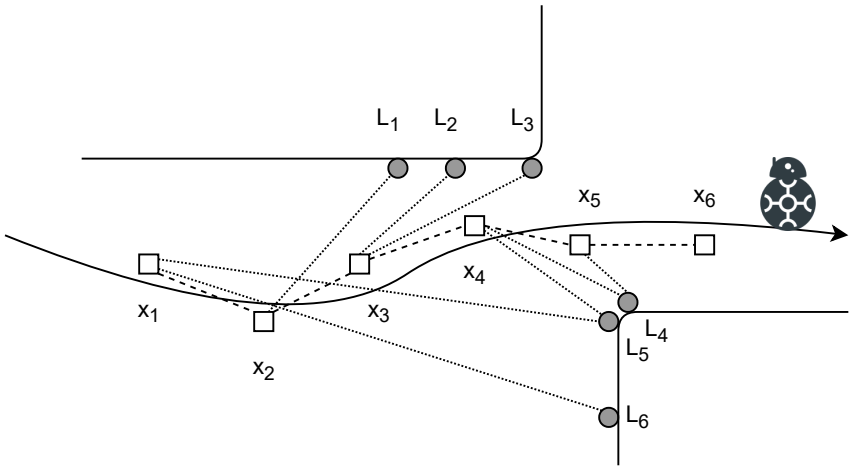


Figure 2.1: Example of a scene (top view) which can be represented in by nodes and edges in a pose graph. The estimated robot poses x_t are indicated by squares, while the estimated landmark positions are indicated by the circles. Constraints for the edges are based on the odometry values (dashed lines) and measurements of the environment (dotted lines).

2.2 Analysis

Practical SLAM implementations end up using a hybrid mix of techniques and may deviate from the pure theoretical algorithms to circumvent issues that arise when using a single specific technique. In this Section, the design and functionality of two real-world SLAM systems will be analysed.

Table 2.1 lists several visual, pose graph based, SLAM approaches of which the source code is published[1]. These systems are often encountered in literature or used as benchmark for comparison to newly developed systems. From the listed SLAM approaches, two are selected to be further analysed in this thesis. This selection is guided by several criteria that are expected to be beneficial for remote visualisation and tele-operation systems:

- Full-SLAM: providing a globally consistent estimation of the current and previously visited positions along the trajectory of the robot.
- Visual/appearance-based: enabling the use of cheap and widely available imaging sensors as primary sensory input and providing information that is useful for 3D reconstruction and visualisation of the environment.
- Pose-graph based: using some form of pose-graph representation for the robot trajectory, facilitating the use of efficient optimisation methods.
- Scale-aware estimation: approaches only using a monocular camera are not able to estimate scale and suffer from scale drift.

RGBiD-SLAM uses a direct image registration method instead of a feature-based approach for tracking camera motion. Although its loop closure detection seems to be feature-based by using a bag-or-words approach, a purely feature-based SLAM approach was preferred. Compared to the other systems, iSAM2 seemed rather old. Furthermore, it was discarded from the selection as it is distributed as part of a larger software framework (GTSAM). Maplab, VINS-Mono and LSD-SLAM all are monocular approaches, which are unable to correct for scale drift. Unfortunately, the source code for the stereo vision S-LSD-SLAM variant (Engel et al. (2015)) is not published.

From the remaining approaches, RTAB-Map and ORB-SLAM2 are selected for further analysis. RTAB-Map seems to gain a lot of attention from within the robotics community. This is a huge benefit when dealing with issues or when adapting RTAB-Map to your application. The KITTI odometry benchmark (Geiger et al. (2012)) is regularly used to test visual odometry and SLAM approaches. An online leader board for comparison of different approaches is also available. Based on this benchmark, ORB-SLAM2 is the best performing among the remaining candidates.

As software in a repository is susceptible to changes, the analysis is based on the specific version at the time of first access. These versions can be identified by the following Git commit hashes:

- RTAB-Map: e57b722ce2d24fff3d1a3a13e5fd008384686304 (29 Oct 2018). This is a commit between the version 0.17.6 and 0.18.0 releases.
- ORB-SLAM2: f2e6f51cdc8d067655d90a78c06261378e07e8f3 (11 Oct 2017).

The selected systems are analysed by investigation of the related literature and source code. The goal of this analysis is to paint a high-level picture of what makes these systems work as they do, which functions are performed and what kind of role these functions play within the whole algorithm. Including an investigation of the source code provides insight into the architectural choices that were made to implement these algorithms.

To achieve these goals, each high-level function that is executed by the SLAM system is analysed separately. This involves distinguishing separated functional steps. Where possible, the separation into functional steps intends to follow the naming and separation as implemented

[1] Note that this does not necessarily mean that the code can be freely used or modified. This depends on the terms of the accompanying license.

Table 2.1: A selection of recent visual, pose graph based SLAM approaches of which the source code is publicly available.

Approach	**Description**
RTAB-Map	Labbé and Michaud (2011).
	https://github.com/introlab/rtabmap.
LSD-SLAM	Engel et al. (2014).
	https://github.com/tum-vision/lsd_slam.
S-PTAM	Pire et al. (2015).
	https://github.com/lrse/sptam.
ORB-SLAM2	Mur-Artal and Tardós (2017).
	https://github.com/raulmur/ORB_SLAM2.
ProSLAM	Schlegel et al. (2018).
	https://gitlab.com/srrg-software/srrg_proslam.
RGBiD-SLAM	Gutierrez-Gomez and Guerrero (2018).
	https://github.com/dangut/RGBiD-SLAM.
iSAM2	Kaess et al. (2011).
	https://github.com/borglab/gtsam.
VINS-Mono	Qin et al. (2018).
	https://github.com/HKUST-Aerial-Robotics/VINS-Mono.
Maplab	Schneider et al. (2018).
	https://github.com/ethz-asl/maplab.

by the authors of the system. Nonetheless, it should be noted that some interpretation was required to include (often minor) behaviour that was not clearly allocated to a step by the original authors. Furthermore, any behaviour that is required for initialisation of the system is ignored.

For each of the functional steps, the analysis consists of three parts:

- A short description of the task that is executed.
- A data-flow diagram showing the interaction between the task at hand and other parts of the system.
- A table which summarises the type of information that is exchanged.

Because information is not only exchanged between different functions, but also stored for later use or to represent a map, storage pools are introduced for both systems. These storage pools are discussed separately when introducing the systems in general. In the data-flow diagrams, the different tasks and storage pools are indicated by the icons shown in Figure 2.2.

Figure 2.2: Icons to be used in the data-flow diagrams to indicate storage elements and functional steps. Storage pools facilitate data storage, while functional steps perform consume, modify or produce data.

As a final note, this analysis attempts to provide high-level insight into the concepts that are of vital importance when implementing a real-world SLAM system. At this level, many low-level implementation details are not relevant to the problem of modularity and reusability. This means that language and platform specific details are mostly neglected, or translated to a more conceptual description. One example of this is the use of the previously mentioned storage pools, which describe both the actual physical memory that is occupied, as well as the bookkeeping and structures to allow access to the stored data.

Another example is the use of synchronisation primitives in a multi-threaded implementation. In this case, the choice was made to ignore mutexes that allow for atomic access and block for a limited amount of time. Behaviour that is more involving, is presented as high-level signalling with status signals such as flags or counters.

2.2.1 RTAB-Map

RTAB-Map, short for "Real-Time Appearance-based mapping", is an approach introduced in Labbé and Michaud (2011). A stereo or RGB-D camera is used as the main source of measurements. The algorithm executes one complete iteration for each image that is fed to the system. Each image is required to be accompanied by an odometry-based pose estimate. This odometry pose must be provided to the core algorithm by an external piece of software. Two visual odometry methods are provided together with RTAB-Map, in case the platform running RTAB-Map does not provide odometry by itself. For the remainder of this thesis, odometry is assumed to be available and the visual odometry methods will not be discussed.

The user can choose between different feature detection techniques and descriptors like SURF, SIFT, FAST, BRIEF and ORB. A bag-of-words approach as introduced by Sivic and Zisserman (2003) is also used, for which the visual words are derived from the previously generated feature descriptors. Using visual words and a bag-of-words for describing the appearance of observations provides an efficient way for RTAB-Map to identify matching features and search for measurements that observe the same scene. The dictionary from which visual words are selected, is constructed dynamically from the feature descriptors that are observed. Similar to the

image feature extraction, a selection of different optimisation methods is provided for the user to choose from. (E.g. based on TORO, g^{2}o or GTSAM)

RTAB-Map appears to be broadly used due to it being freely available for a broad range of platforms. It comes with additional tools for visualisation of the mapped environment and integrates easily with ROS. Therefore, it provides a lot of functionality and configurability and integrates well as a stand-alone application. It is written in C++ and comes under a BSD 3-clause license. At the time of writing, RTAB-Map is actively maintained but the core algorithm does not seem to be under development or receiving big changes.

Looking at the source code, the core functionality is contained within the corelib folder of the repository. Most of the SLAM algorithm logic is contained within two large files: Rtabmap.cpp and Memory.cpp, with other files containing supporting code. Unfortunately, documentation about the software architecture and functionality contained in these files is not publicly available. In the context of this analysis, execution of the function *process()* in Rtabmap.cpp is considered one iteration of the algorithm.

RTAB-Maps key feature is a memory management approach, which divides its data into three storage pools: short-term, working and long-term memory. This separation fulfils two separate goals:

- Recent data is kept in short-term memory to avoid acting on data that is likely to be similar to other recent measurements.
- The computational load is kept within bounds by only using data that is available in working memory and excluding data that is stored in long-term memory.

Within these three storage pools, data is stored in the form of objects called Signatures. A Signature aggregates all information that is related to an observation made at a specific moment in time. This involves static data that is sampled once: the control u_t, measurement z_t and other constants at the time of sampling. But also includes values that are updated during the execution of the SLAM algorithm, such as weights, status flags and connections to other Signatures.

Connections between Signatures in storage are made with several types of Links. The type of a Link indicates the different sources in the algorithm from which the Links originate, such as the loop closure detection. Links define a spatial relation between two Signatures and contain a transformation which describes the displacement from the one Signature to the other. The information contained by Links is used throughout the algorithm, for example to identify neighbouring Signatures in the trajectory followed by the robot. The constraints for the pose graph optimisation are also generated from the transformations within the Links. Together with the raw odometry poses, this yields the trajectory estimate $x_{1:t}$ after optimisation.

Table 2.2 lists the steps for one iteration of RTAB-Maps algorithm, in the order of their execution and with a short description of each task that is performed. Each of these steps is further analysed in this Section, in order to identify the corresponding data dependencies, interactions with other functions in the system and the type of information that is exchanged. For each step, the accompanying figures and tables summarise the data dependencies and provide details about the type of information that is exchanged via these dependencies. Figure 2.10 at the end of this Section presents an overview of the steps and dependencies that are discussed.

It should be noted that some functionality present in the source code was ignored during the analysis of the behaviour and dependencies, in order to focus on what are assumed to be the most important tasks that are performed. The majority of the ignored behaviour is already disabled in the default configuration of RTAB-Map. So called "Intermediate nodes" are often encountered in the code and represent Signatures without sensor data, but with an estimated

pose. These Signatures are only used as instrumentation for some benchmarks and are therefore ignored. Furthermore, behaviour related to GPS data, laser scan data, path planning or the localisation mode is also not covered in this thesis.

Table 2.2: Overview of functional steps for one iteration of RTAB-Map, in order of execution.

	Step	Description
1.	Create Signature	Creating and storing a sample containing measurement (features, descriptors, visual words) and control (odometry pose) for further use by the algorithm.
2.	Rehearsal	Reduce data in storage if the current observation is very similar to the previous observation.
		Enable data to be used for evaluation during loop closure detection.
3.	Bayes filter	Find observations from the past which appear sufficiently similar to the current observation to form a loop closure.
4.	Retrieval	Retrieve existing data that may be related to the robots current location, for use by the algorithm.
5.	Loop closure Link	Determine a constraint between two observations that are accepted by the Bayes filter as a loop closure.
6.	Pose graph optimisation	Estimate the trajectory of robot poses that minimises the error with respect to the constraints that are available in the data set.
7.	Transfer	Limit the execution time of the algorithm by reducing the data set that is used by the algorithm.

Create Signature

The first step in RTAB-Map is the creation of a Signature object for use throughout the entire system. As mentioned before, a camera image together with an odometry pose is provided to the system for each iteration of the algorithm. The odometry pose (from u_t)) is stored in the Signature and represents the first estimate of the robot position at the current time. The pose is expected to be an absolute position with respect to the origin of the odometry reference frame, which is situated at the location where the system was started and initialised. This odometry pose is never changed afterwards, as RTAB-Map keeps track of the optimised pose estimates separately.

Using the odometry poses of the current and previous Signatures, the first Link is created between the new (current) Signature and the previous Signature in short-term memory. This Link is also stored in both involved Signatures and is used to keep track of the path that is followed. The transformation associated with this Link is identical to the difference between both associated odometry poses.

The visual information from the camera image is immediately used to extract the required image features, descriptors and visual words during this step. The image itself is not used by the

algorithm any further, but RTAB-Map does store the image for future reference within the Signature. Because of this, the measurement (z_t) that is used by the algorithm consists of the image features (with 2D and 3D coordinates), descriptors and visual words that are part of the bag-of-words for this Signature.

Figure 2.3 shows how Create Signature exchanges information with other parts of the system, via dependencies on the short-term memory, pose graph optimisation and Transfer. These dependencies are also listed in Table 2.3, which summarises the information that is exchanged.

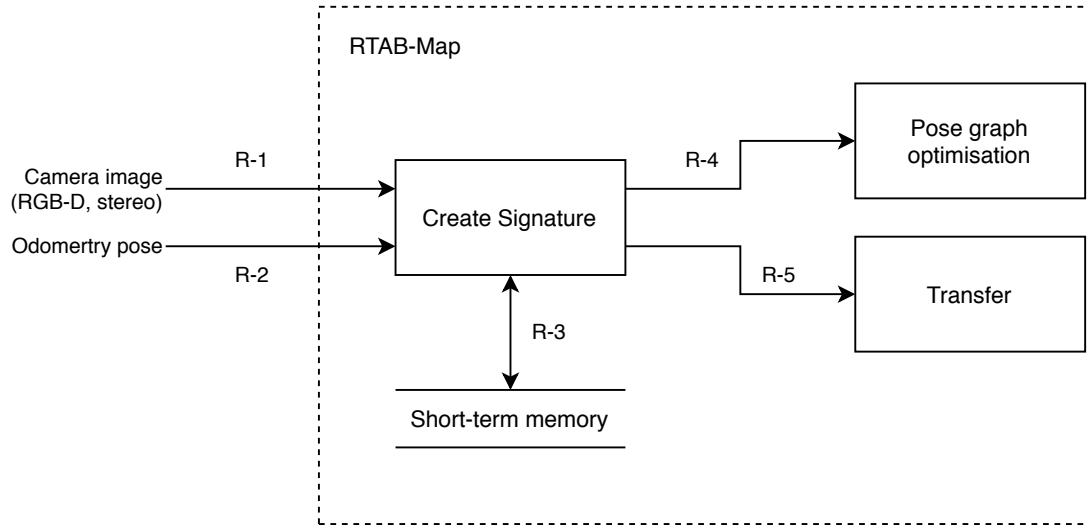


Figure 2.3: Exchange of information and interaction between the Create Signature functionality of RTAB-Map and other parts of the system.

Table 2.3: Data dependencies for the Create Signature step in RTAB-Map.

Dependency	Direction	Description
R-1	In	Appearance (RGB or grey scale colour information) and structure (pixel depth or stereo image). Spatially sampled as pixels.
R-2	In	6 degrees of freedom odometry pose. Accompanied by a covariance matrix as a measure of uncertainty.
R-3	In	Data related to the previous Signature, such that the newly created Signature can be linked to it.
R-3	Out	Storing the newly created Signature and its Link to the previous Signature.
R-4	Out	Flag: information in storage has been modified/updated.
R-5	Out	Number: one Signature has been added to storage.

Rehearsal

The second step that can be identified is called Rehearsal. This step aims to reduce the data set on which the algorithm operates. If the current observation is visually very similar to the previous one, the new Signature and the data it contains is discarded. Along with the inform-

ation that is part of the observation that is made, each Signature contains a weight. Rehearsal increases the weight that is associated with the previous Signature to register such an event. An increased weight can be regarded as a measure of confidence, as it indicates that a Signature is supported by multiple similar observations.

Figure 2.4 shows Rehearsal and its dependencies to other parts of RTAB-Map. Fulfilling the task requires a dependency on short-term memory to access the required information, while the pose graph optimisation and Transfer are notified about the result. A summary of the information that is exchanged over these dependencies is listed in Table 2.4.

The dependency between the short-term and working memory is not directly related to Rehearsal, but happens immediately after Rehearsal. If the size of the short-term memory grows over a specified limit, the oldest Signature in short-term memory is moved to the working memory. This seems to be a minor step, but plays an important role within RTAB-Map. Keeping recent observations in short-term memory excludes them from loop closure detection at a later stage. Once the information has matured, the move to working memory allows for inclusion during loop closure detection.

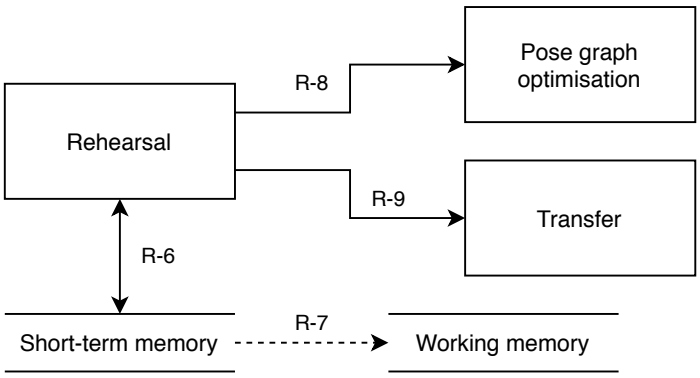


Figure 2.4: Exchange of information and interaction between the Rehearsal functionality of RTAB-Map and other parts of the system.

Table 2.4: Data dependencies for the Rehearsal step in RTAB-Map.

Dependency	Direction	Description
R-6	In	Data related to the current and previous Signature for comparison of both observations.
R-6	Out	In case of Rehearsal: deleting the current Signature and updating the weight of the previous Signature.
R-7	-	Post-Rehearsal: move oldest Signature in short-term memory to working memory.
R-8	Out	Flag: information in storage has been modified/updated.
R-9	Out	Number: one Signature has been deleted from storage.

Bayes filter

The Bayes filter step that is executed next is the first part of the loop closure detection in RTAB-Map. During this step, the likelihood of a loop closure existing between the current Signature

(in short-term memory) and previous observations in working memory is estimated. The prediction is based on a direct comparison of the visual appearance of image features in the observations that are involved. It should be noted that no spatial information about the location of image features is considered, which means that the structure of the observed scene is ignored during this prediction step.

RTAB-Map follows an approach by Adrien Angeli and Meyer (2008), for which the bag-of-words (Sivic and Zisserman (2003)) of each Signature and associated scoring mechanism is used as a metric for comparing the observations. A virtual Signature represents the most commonly observed visual words. This virtual Signature is used to adjust the hypothesis, such that the prediction is based on sufficiently unique words.

The Bayes filter operation produces two results that feed to the Retrieval and Loop closure Link steps. The first part of the result is which Signature in working memory has the highest probability of forming a loop closure with the current Signature. The second part of the result is the decision by the Bayes filter step if the hypothesis is sufficiently strong to be accepted as an actual loop closure. These interactions are shown in Figure 2.5. Table 2.5 summarises the information that is exchanged.

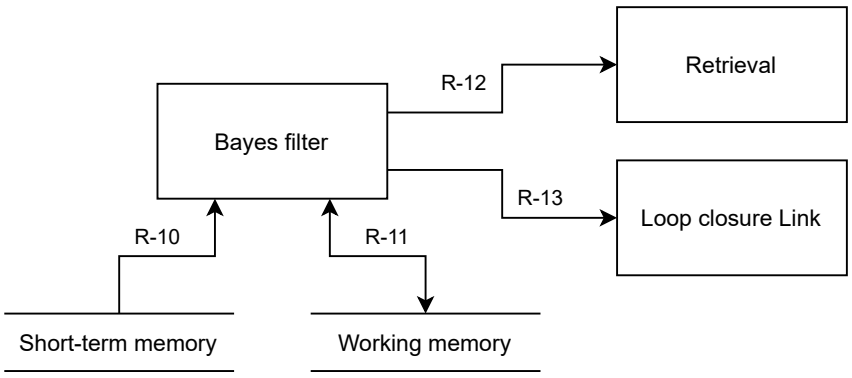


Figure 2.5: Exchange of information and interaction between the Bayes filter functionality of RTAB-Map and other parts of the system.

Table 2.5: Data dependencies for the Bayes filter step in RTAB-Map.

Dependency	Direction	Description
R-10	In	Data related to the current Signature for comparison of the visual appearance of the observation.
R-11	In	Data related to all working memory Signatures for comparison of the visual appearance against the current Signature.
R-11	Out	Storing the virtual Signature.
R-12	Out	This Signature in working memory has the highest probability of forming a loop closure with the current Signature.
R-13	Out	This Signature in working memory forms a loop closure with the current Signature.

Retrieval

Signatures may be stored in the long-term memory for reasons that will be discussed at a later stage. Retrieval is tasked with fetching data from the long-term memory back into working memory. Retrieving this data back into working memory makes it available for use by the algorithm, for example during the Bayes filter step.

The strongest hypothesis from the Bayes filter is used as a starting point for the selection of Signatures that may contain valuable information, along with several other criteria. Figure 2.6 shows that Retrieval interacts with the short-term, working and long-term memory to fulfil its task. The behaviour of the pose graph optimisation and Transfer steps is also affected by the outcome of Retrieval. Table 2.6 provides details about these dependencies.

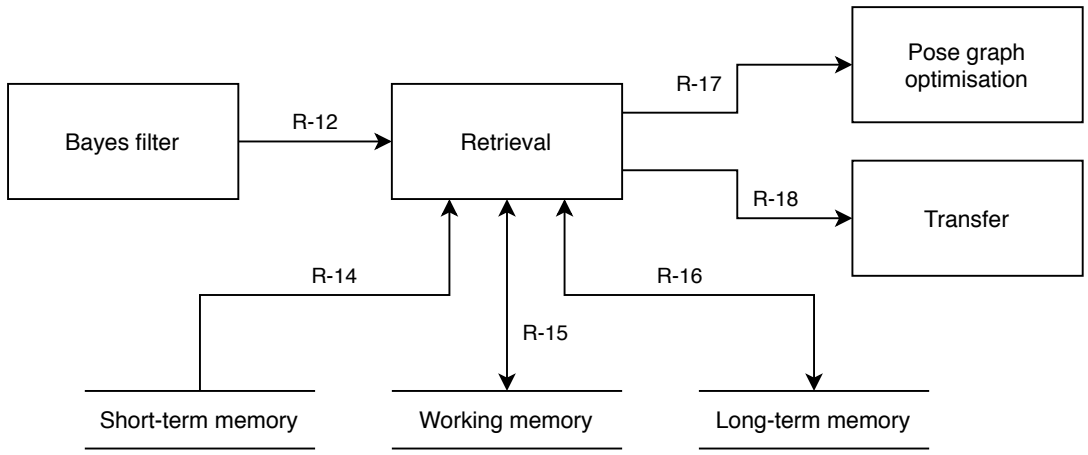


Figure 2.6: Exchange of information and interaction between the Retrieval functionality of RTAB-Map and other parts of the system.

Table 2.6: Data dependencies for the Retrieval step in RTAB-Map.

Dependency	Direction	Description
R-12	In	This Signature in working memory has the highest probability of forming a loop closure with the current Signature.
R-14	In	Data related to Signatures in short-term memory used to select potentially valuable Signatures from long-term memory to be retrieved.
R-15	In	Data related to Signatures in working memory used to select potentially valuable Signatures from long-term memory to be retrieved.
R-15	Out	Storing Signatures that are retrieved from long-term memory.
R-16	In	Data related to Signatures in long-term memory used for selection and retrieval of potentially valuable Signatures.
R-16	Out	Deleting Signatures that are retrieved from long-term memory.
R-17	Out	Flag: information in storage has been modified/updated.
R-18	Out	*1)* Number: amount of Signatures retrieved. *2)* A list of Signatures that have to be excluded from being transferred to long-term memory.

Loop closure Link

The next step performs the second part of the loop closure detection in RTAB-Map: generation of the loop closure Link between two Signatures. Such a loop closure Link is generated between the Signature that is indicated by the Bayes filter step and the current Signature.

The constraint associated to this Link provides an estimated displacement and rotation between both observations. This estimate follows from the difference between the location of image features observed by both Signatures that are involved with the loop closure. Corresponding points in both images can be related to a certain displacement between the robot poses when taking both observations. While determining the transformation that describes the associated constraint, this step also validates the loop closure at the same time.

Figure 2.7 shows the loop closure Link creation step and its interactions with other parts of the system. Details about the information that is exchanged is listed in Table 2.7.

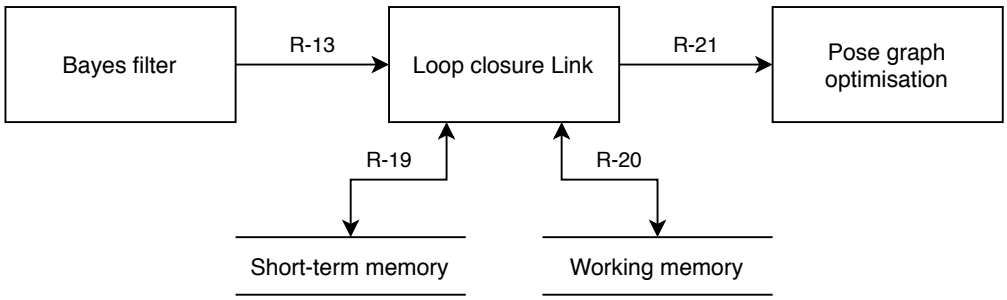


Figure 2.7: Exchange of information and interaction between the loop closure Link functionality of RTAB-Map and other parts of the system.

Table 2.7: Data dependencies for the loop closure Link step in RTAB-Map.

Dependency	Direction	Description
R-13	In	This Signature in working memory forms a loop closure with the current Signature.
R-19	In	Data related to the current Signature used to determine the constraint between both observations associated with the loop closure.
R-19	Out	Storing the Link that describes the loop closure constraint between both involved Signatures.
R-20	In	Data related to the loop closure candidate Signature used to determine the constraint between both observations associated with the loop closure.
R-20	Out	Storing the Link that describes the loop closure constraint between both involved Signatures.
R-21	Out	Flag: information in storage has been modified/updated.

Pose graph optimisation

Any changes by the previously executed steps provides new information which may improve the estimated trajectory. Pose graph optimisation is performed in order to update the estimation with the most recent information. Figure 2.8 depicts this behaviour by the dependencies that are shown at the left side, which trigger an optimisation step.

The pose graph for optimisation is constructed from the information contained in both short-term and working memory. After optimisation has finished, the result is validated before saving it for the next iteration. If the result does not pass this validation, it is discarded and any new loop closure Links involved are deleted.

The optimised pose estimates for the Signatures are stored separately from the original odometry poses. Dependency R-22 and R-23 as listed in Table 2.8 are not completely true to the actual implementation, as the optimised poses are not fed back into the short-term and working memories. To document the behaviour accurately, a fourth storage pool should be defined for RTAB-Map. Because this would make the figures for RTAB-Map a lot more complicated and the optimised pose storage is kept synchronously with short-term and working memory, the

choice was made to ignore this specific separation. Unfortunately, RTAB-Map's implementation requires a lot of additional bookkeeping to keep the optimised poses synchronised with contents of the short-term and working memories.

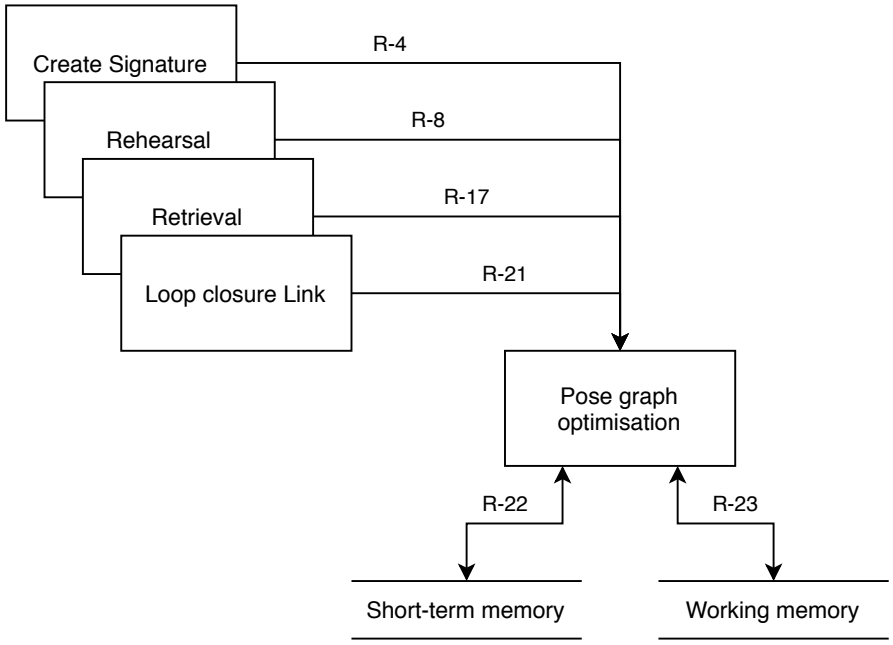


Figure 2.8: Exchange of information and interaction between the pose graph optimisation functionality of RTAB-Map and other parts of the system.

Table 2.8: Data dependencies for the pose graph optimisation step in RTAB-Map.

Dependency	**Direction**	**Description**
R-4	In	Flag: information in storage has been modified/updated by the Create Signature step.
R-8	In	Flag: information in storage has been modified/updated by the Rehearsal step.
R-17	In	Flag: information in storage has been modified/updated by the Retrieval step.
R-21	In	Flag: information in storage has been modified/updated by loop closure Link step.
R-22	In	Data related to all short-term memory Signatures to construct a pose graph, pre-optimisation.
R-22	Out	Storing optimised poses, post-optimisation.
R-23	In	Data related to all working memory Signatures to construct a pose graph, pre-optimisation.
R-23	Out	Storing optimised poses, post-optimisation.

Transfer

The last step during an iteration of RTAB-Map is called Transfer, which is the opposite of Retrieval. Transfer is tasked with moving information from working to long-term memory whenever the execution time for one iteration or the working memory size exceeds a specified threshold. Data in the long-term memory is excluded from use by the next iteration of the algorithm, reducing the computational load.

Additional to the dependencies that follow from the description above, Figure 2.9 also shows that there are interactions with other parts of the system. These signals are part of the process that selects candidate Signatures for being Transferred. As before, Table 2.9 provides more details about the information that is exchanged.

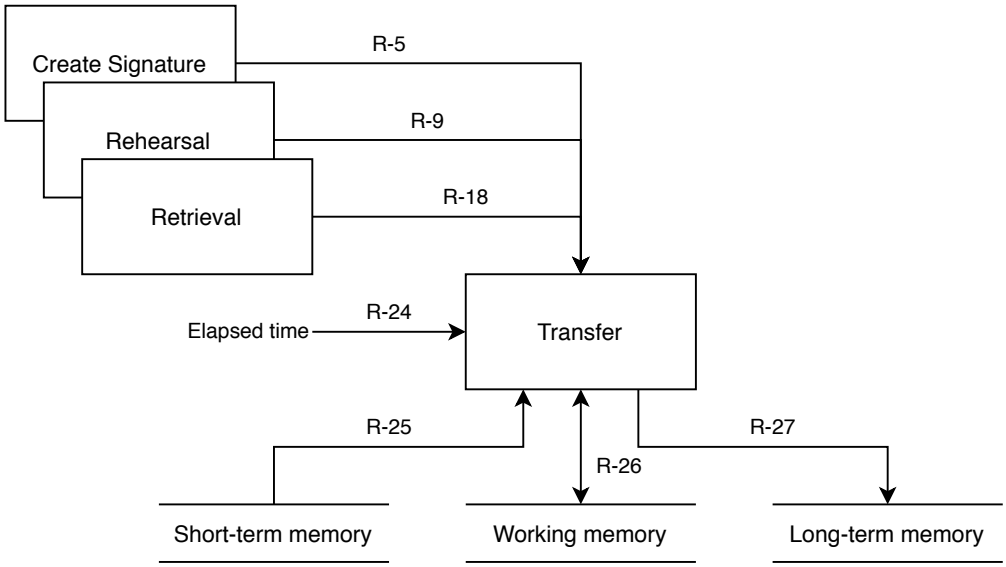


Figure 2.9: Exchange of information and interaction between the Transfer functionality of RTAB-Map and other parts of the system.

Table 2.9: Data dependencies for the Transfer step in RTAB-Map.

Dependency	Direction	Description
R-5	In	Number: one Signature has been added to storage.
R-9	In	Number: one Signature has been deleted from storage.
R-18	In	*1)* Number: amount of Signatures retrieved. *2)* A list of Signatures that have to be excluded from being transferred to long-term memory.
R-24	In	Elapsed time since the start of the current iteration up to Transfer.
R-25	In	Data related to short-term memory Signatures used during the selection of Signatures from working memory for transfer.
R-26	In	Data related to working memory Signatures for the selection and transfer of Signatures to long-term memory.
R-26	Out	Deleting Signatures that are transferred to long-term memory.
R-27	Out	Storing Signatures that are transferred to long-term memory.

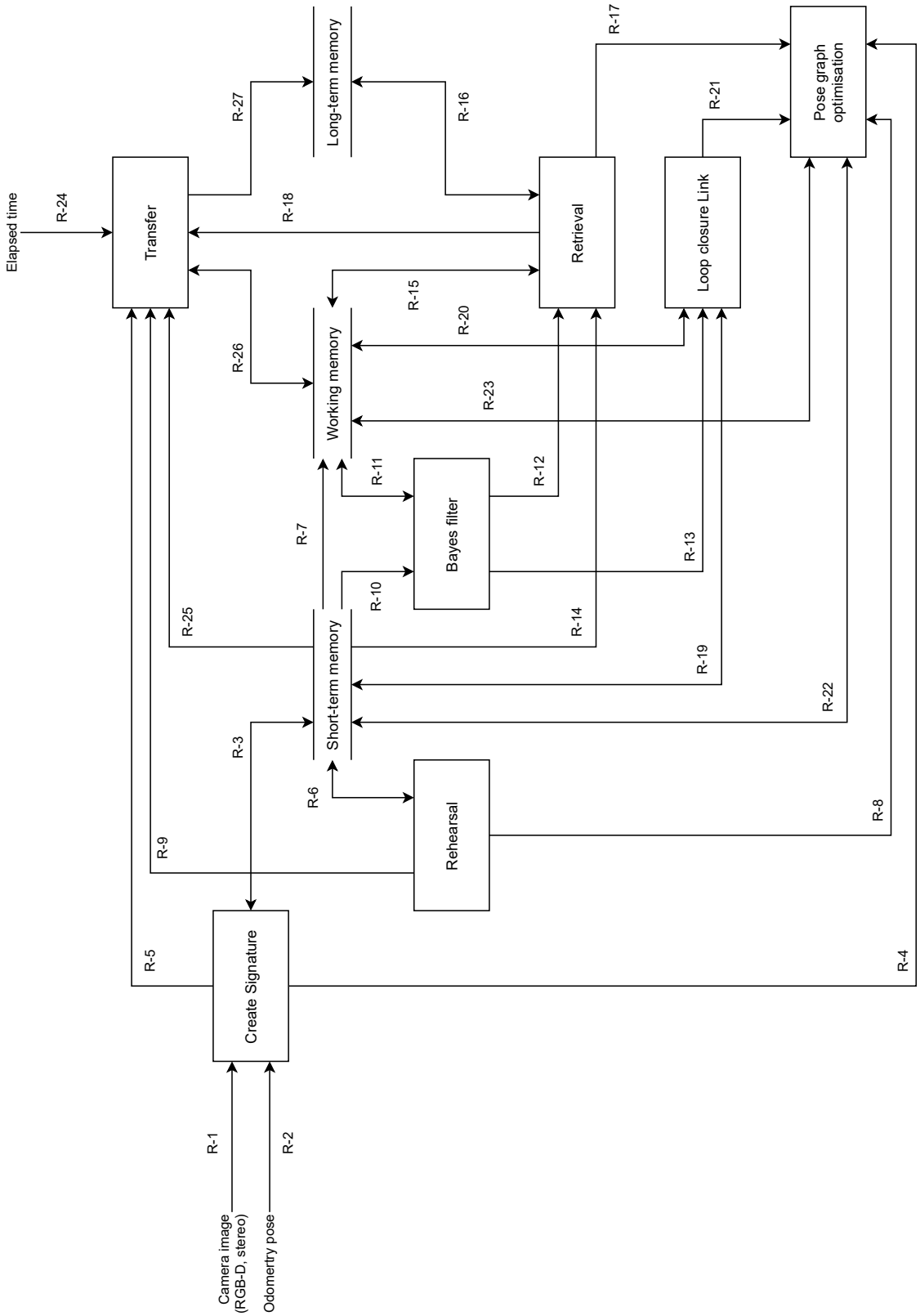


Figure 2.10: System overview of RTAB-Map showing data dependencies between the identified functional steps.

2.2.2 ORB-SLAM2

ORB-SLAM2 (Mur-Artal and Tardós (2017)) ads RGB-D and stereo vision support to its monocular predecessor ORB-SLAM (Raúl Mur-Artal (2015)). It is a feature-based system for performing real-time SLAM with loop closure detection and the possibility to re-locate itself within a given map. The only input to ORB-SLAM2 is an RGB-D or stereo image from a camera sensor. As the name implies, Oriented FAST and Rotated BRIEF (ORB) features are used throughout the system. Visual words for a bag-of-words (Gálvez-López and Tardós (2012), which is based on Sivic and Zisserman (2003) and Adrien Angeli and Meyer (2008)) are also used for efficient appearance-based comparison and matching for loop closure and landmark detection. The g^2o framework (Rainer Kümmerle (2011)) is used to perform several bundle adjustment and pose graph optimisations.

ORB-SLAM2 is distributed including several benchmark examples and code to integrate with ROS. A map viewer is also included. It is written in C++11 and comes with a GPLv3 license. At the time of writing, ORB-SLAM2 does not seem to be actively maintained.

The algorithm is divided over multiple threads that (partially) operate asynchronously. Only the so-called Tracking thread runs at the cameras frame rate. The other threads are performing what is called Local Mapping, Loop Closing and the global/full bundle adjustment optimisation. These are all triggered by events that mark the availability of updated information relevant to each function.

ORB-SLAM2 uses three types of objects to manage and arrange information throughout the system:

- Frames
- Key Frames
- Map Points

Frames and Key Frames aggregate the information that is available with respect to a single observation of the environment by the robot, taken at a single moment in time. This involves static data that is sampled once: the measurement z_t and other constants at the time of sampling. But also includes values that are updated during the execution of the SLAM algorithm, such as statistics, status flags and connections to other objects like Frames, Key Frames and Map Points. Frames are only relevant for the functionality within the Tracking thread and are created for each camera image. Key Frames are derived from Frames from time to time at a reduced rate and are primarily used for the mapping process and landmark detection.

Each detected landmark is represented by a Map Point, which aggregates all associated information, such as the estimated pose and the Key Frames by which it is observed. Landmark observations are registered bidirectionally, by storing the associated references both within the affected Key Frame and Map Point. This provides a mechanism to traverse along observations in both directions: by selecting Map Points via an observing Key Frame, or by selecting Key Frames via a specific Map Point.

Key Frames are also linked together by two mechanisms: the Spanning Tree and the Co-visibility Graph. The Spanning Tree links Key Frames together in the order in which they are created. It can be traversed in both directions, providing the means to iterate along all Key Frames known to the system. The Co-visibility Graph is constructed from observations of landmarks by all Key Frames. When two Key Frames observe the same landmark, these are said to be co-visible. These Key Frames are connected in the Co-visibility Graph. Each connection in the Co-visibility Graph describes how many landmarks are shared between those Key Frames.

The resulting graph is used throughout the system as a mechanism to select neighbouring Key Frames based on the number of shared observations.

ORB-SLAM2s bag-of-words approach makes use of a static dictionary which assigns weights to specific words. These weights are correlated to the uniqueness of visual words, based on their occurrence within a reference data set of images. A scoring mechanism is also provided based on these weights, allowing for a direct comparison of two bags-of-words.

For the sake of this analysis, three storage pools can be identified in ORB-SLAM2: Tracking, Global Map and the Key Frame database.

- The Tracking storage pool is not formally defined by the authors, but can be identified within the software as the data that is only accessed by the functions of the Tracking thread.
- The name Global Map will be used to refer to what the authors are calling just Map, to prevent confusion with the general term map. This Global Map stores the data related to all Key Frames and Map Points created by ORB-SLAM2. Each Key Frame also contains fields to store the data which constitute the Co-visibility Graph and Spanning Tree in a distributed way.
- The Key Frame database (name from code) or Recognition database (name from papers) stores the information that is required to perform place recognition. This information is limited to a registry such that Key Frames in the Global Map can be found by a search for observed visual words in their bag-of-words.

Each of the previously mentioned objects (Frames, Key Frames and Map Points) is associated with a position in the environment.

By using the Spanning Tree, the trajectory $x_{1:t}$ can be reconstructed from the poses of Key Frames. The pose of a Key Frame always represents the most recent estimate available to the system. The space between Key Frames in the trajectory is filled in by using the poses associated with Frames. Because Frames are not stored for future reference, this is done by separately saving each Frame's pose relative to its reference Key Frame. The actual pose of a Frame can then be recovered by using the relative pose and the current estimate for the corresponding reference Key Frame.

Each pose of a Map Point is directly contributing to the map m. When a new Map Point is created, the position of the landmark directly follows from the projection of the associated image feature into the world, using the pose of the observing Frame or Key Frame. Later, the pose is adjusted by combining multiple observations in a bundle adjustment optimisation.

Table 2.10 lists the steps of the ORB-SLAM2 algorithm that will be discussed in the following sections. Due to the implementation using asynchronous threads, these steps are not necessarily always executed in the order they are listed. For each of the steps, a short description of the task that is performed is included. At the end of this Section, an overview of the identified functional steps and dependencies is given in Figure 2.21.

Because of the scope defined at the beginning of Section 2.2, behaviour that is related to ORB-SLAM2's monocular SLAM mode and the so-called Localisation mode is not covered. The Full Bundle Adjustment is also ignored, as this operation is rather well separated from the other parts of the algorithm. Furthermore, Raúl Mur-Artal (2015) mentions that *"when performing the pose graph optimization, the solution is so accurate that an additional full bundle adjustment optimization barely improves the solution"*.

Table 2.10: Overview of functional steps for processing an image in ORB-SLAM2.

	Step	Description
1.	Create Frame	Creating and storing a sample containing the measurement (features, descriptors, visual words) for further use by the algorithm.
2a.	Track with motion model	Estimate current pose.
2b.	Track reference Key Frame	Estimate current pose.
2c.	Re-localisation	Estimate current pose.
3	Track Local Map	Refine estimation of current pose.
4	Create Key Frame	Decision making for and creation of a new Key Frame out of the current Frame.
5	Process Key Frame	Incorporating a newly created Key Frame into the Global Map.
6	Local Mapping	Data association for landmark detection and reduction of data contained within the Global Map.
7	Loop detection	Detecting loop closures and estimation of the transformation describing the constraint between the two involved observations.
8	Loop correction	Inserting the detected loop closure constraint into the map and pose graph optimisation to update the estimated trajectory and landmark poses.

Create Frame

The first step in ORB-SLAM2 is creating a Frame object, which represents the current observation. At a later stage, the corresponding estimated robot pose is also added to the Frame. During this first step, the visual and structural information is extracted from the stereo or RGB-D image and stored in a generalised format for use by the rest of the system. Visual information consists of ORB features and corresponding descriptors, as well as the bag-of-words which may be derived from the descriptors at a later stage.

The generalised format provided by a Frame consists of both left and right image features, as well as the corresponding depth for matching features. For RGB-D input, the position of the "virtual" features in the right image are determined using the depth information. For stereo image input this is the other way around, by determining the depth from the available matched stereo features.

Colour information is discard by converting image data to grey scale before performing feature extraction. The original image is not stored for further use by ORB-SLAM2.

Figure 2.11 shows how Create Frame interacts with other parts of the system. The information that is exchanged over the corresponding interfaces is listed in Table 2.11.

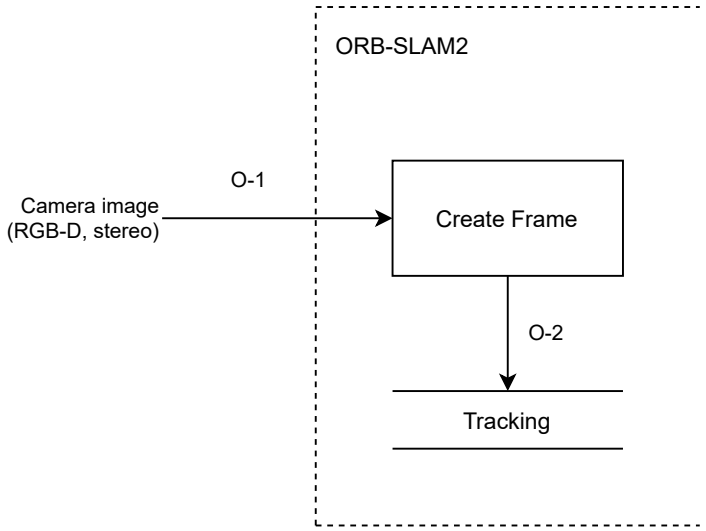


Figure 2.11: Exchange of information and interaction between the Create Frame functionality of ORB-SLAM2 and other parts of the system.

Table 2.11: Data dependencies for the Create Frame step in ORB-SLAM2.

Dependency	Direction	Description
O-1	In	Appearance (RGB or grey scale colour information) and structure (depth image or via stereo image). Spatially sampled as pixels.
O-2	Out	Storing the newly created Frame.

Track with motion model

During regular operation, the Track with motion model step provides the first estimate for the current robot pose. This step performs visual odometry by matching image features of the current Frame against those of the previous Frame. Although Frames are not used to formally register observations of landmarks in the Global Map, previously matched Map Points are used to select image features for matching against the current Frame. ORB-SLAM2 uses the absolute pose that is associated with each Map Point for re-projecting these image features.

Because the matching for visual odometry is based on absolute poses of features, the current robot pose should be estimated first. This guess is based on the pose of the previous Frame and a motion model which takes the robots movement into account. An estimate for the current velocity is provided by the Track Local Map step, calculated during processing of the previous Frame. Once sufficient matches are found, a bundle adjustment optimisation finalises the pose estimation for the current Frame.

Figure 2.12 shows the expected dependency on the Tracking storage for access to the previous Frame. An additional dependency on the Global Map is also required to access information about the position of landmarks and update the previous Frame pose if its reference Key Frame is updated due to Local Mapping or loop closure detection. Table 2.12 summarises the information that is exchanged between the dependencies.

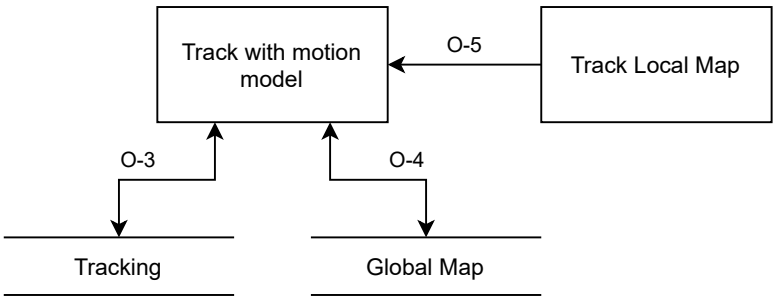


Figure 2.12: Exchange of information and interaction between the Track with motion model functionality of ORB-SLAM2 and other parts of the system.

Table 2.12: Data dependencies for the Track with motion model step in ORB-SLAM2.

Dependency	Direction	Description
O-3	In	Data related to the current and previous Frames, such that the current pose can be estimated and image features can be matched.
O-3	Out	Storing updated pose estimates for the current and previous Frames, as well as matched image features in the current Frame.
O-4	In	Most recent pose estimate of the reference Key Frame and relevant Map Points.
O-4	Out	Storing status information within Map Points that are discarded as outliers.
O-5	In	Estimated current velocity for use by the motion model.

Track reference Key Frame

In some cases, Track with motion model may not be able to provide a pose estimate for the current Frame. Track reference Key Frame provides an alternative method to determine the current pose of the robot. As the name implies, the observation of the current Frame is compared against the reference Key Frame. This reference Key Frame provides a pose and a set of landmarks that are used for matching against image features in the current Frame.

Matching is also performed in an alternative way, using visual words out of the bag-of-words of the Key Frame. Therefore, derivation of the bag-of-words for the current Frame is also performed.

Figure 2.13 indicates where information is exchanged, while Table 2.13 summarises the data that is exchanged.

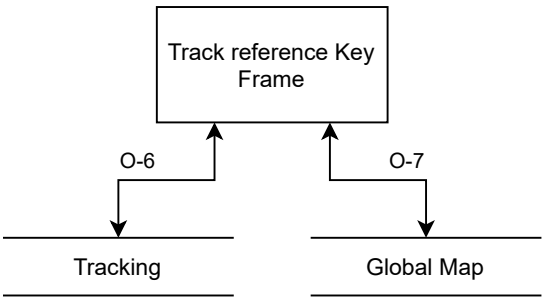


Figure 2.13: Exchange of information and interaction between the Track reference Key Frame functionality of ORB-SLAM2 and other parts of the system.

Table 2.13: Data dependencies for the Track reference Key Frame step in ORB-SLAM2.

Dependency	Direction	Description
O-6	In	Data related to the current Frame for derivation of the bag-of-words. Previous Frame pose.
O-6	Out	Storage of the pose estimate, matched landmarks and bag-of-words for the current Frame.
O-7	In	Data related to the reference Key Frame and associated Map Points.
O-7	Out	Storing status information within Map Points that are discarded as outliers.

Re-localisation

If a Frame cannot be successfully matched against previous observations for the tracking steps explained above, ORB-SLAM2 becomes lost. Once this happens, the mapping process stops due to it not being fed with new information and the current robot pose is unknown.

To get back to normal operation, new Frames are now used for the Re-localisation step, which determines the current robot position within the Global Map. This is achieved by performing place recognition between the current Frame and all Key Frames in the Global Map. A bag-of-words approach is used to find candidate Key Frames that observe a scene with a similar appearance as the current Frame. Image features of candidate Key Frames are compared to the current Frame by using the corresponding ORB descriptors. The matched features are then used to estimate the current robot pose with respect to the associated Key Frame.

Figure 2.14 shows that Re-localisation depends on information out of each of the three storage pools within ORB-SLAM2. More details are provided in Table 2.14. If Re-localisation successfully determines the current robot pose, ORB-SLAM2 returns to normal tracking operation. Otherwise, the system remains lost and continues to process Frames for Re-localisation.

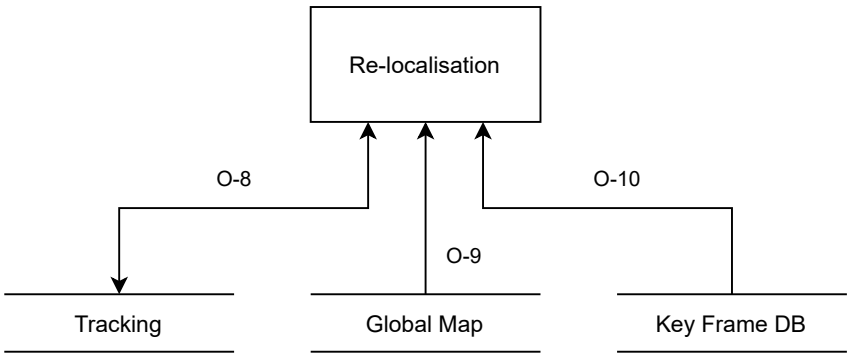


Figure 2.14: Exchange of information and interaction between the Re-localisation functionality of ORB-SLAM2 and other parts of the system.

Table 2.14: Data dependencies for the Re-localisation step in ORB-SLAM2.

Dependency	Direction	Description
O-8	In	Data related to the current Frame used to derive the bag-of-words and perform matching of image features.
O-8	Out	Storing the bag-of-words, matched Map Points, outlier Map Points and estimated pose for the current Frame.
O-9	In	Data related to the selected candidate Key Frames for matching of image features.
O-10	In	Bag-of-words lookup to identify candidate Key Frames that observe features with a similar appearance as the current Frame.

Track Local Map

The Track Local Map step follows after an initial pose estimate has been determined by Track with motion model, Track reference Key Frame or Re-localisation. Each of these three steps operated on a limited data set, which directly relates to the Frame or Key Frame that was used as reference input. Along with the pose estimate, these steps also found matches between feature of the current Frame and some landmarks.

Using the estimated pose and matched landmark observations by those previous steps, an extended data-set is sampled from the Global Map. This sub-set of the Global Map is called the "Local Map" and contains Key Frames and Map Points that are selected following several criteria, involving the Co-visibility Graph and Spanning Tree. It should be noted that this Local Map for Tracking has no direct relation to the data-set on which the Local Mapping step is operating. Both steps use different criteria and mechanisms for the selection of the data they work with.

The landmarks contained in the Local Map are matched against image features of the current Frame that are not yet associated with a Map Point. This should provide an increased number of image features that is associated with a landmark. A bundle adjustment operation updates the estimated robot pose, marking any outlier matches in the process.

The Track Local Map step also updates the so-called Found Ratio, a statistic that is stored along Map Points in the Global Map. It describes the ratio between how often a Map Point is expected

to show up within the image of a Frame (by re-projection) and how often the same Map Point is successfully matched without being rejected by the bundle adjustment. The Found Ratio is cumulative and therefore averages over multiple Frames as long as the corresponding Map Point is within the view of the camera.

After Track Local Map has completed, the updated pose estimate is used to determine the current velocity at which the robot is moving. This velocity is used by the motion model, feeding back information to the Track with motion model step.

These dependencies are shown in Figure 2.15 and summarised in Table 2.15.

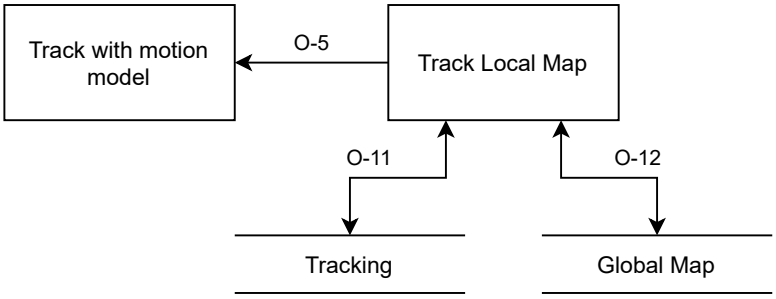


Figure 2.15: Exchange of information and interaction between the Track Local Map functionality of ORB-SLAM2 and other parts of the system.

Table 2.15: Data dependencies for the Track Local Map step in ORB-SLAM2.

Dependency	Direction	Description
O-5	Out	Motion model velocity.
O-11	In	Data related to current Frame required for the construction of the Local Map and for matching of features to landmarks.
O-11	Out	Storing data related to the current Frame, such as updated pose estimate, matched Map Points and outliers.
O-12	In	Data related to all Key Frames to select observed Map Points that are relevant to the current Frame. Data related to these Map Points for matching against features in the current Frame and during pose estimation.
O-12	Out	Storing statistics and re-projected coordinates for observations of Map Points by the current Frame.

Create Key Frame

The Create Key Frame step draws the border between the part of the algorithm that operates on each Frame and the remainder that is running at a lower rate. In order to decide for the creation of a Key Frame, several criteria are monitored during this step. Once these criteria are met, a new Key Frame is created out of the Frame that is currently processed by the Tracking thread.

During the creation of a Key Frame, the depth associated with detected image features is evaluated. All features of which the depth is below a certain threshold are "close" and therefore

required to be associated with a landmark. If during the previous operations no existing match was detected, a new Map Point is created. Image features that are "far" may be evaluated in a similar way if there are insufficient close points.

As indicated by Figure 2.16, ORB-SLAM2 directly stores information in the Global Map during this step. It should be noted that this is not yet the case for the newly created Key Frame. For the newly created Map Points, the existence of the observation is immediately added to the Global Map, by registering the observation in the affected Key Frame and Map Point objects. This allows for the previously described steps in the Tracking thread to directly make use of these new landmarks. Any previously existing Map Points were already known to the Tracking thread, such that registration of the observation by the Key Frame can be postponed. Therefore, the new Key Frame is handed off to the Local Mapping thread, which takes care of adding it to the Global Map and registering the observation of previously existing Map Points. This will be discussed in the next Section. Table 2.16 indicates the flow of information across the different interfaces.

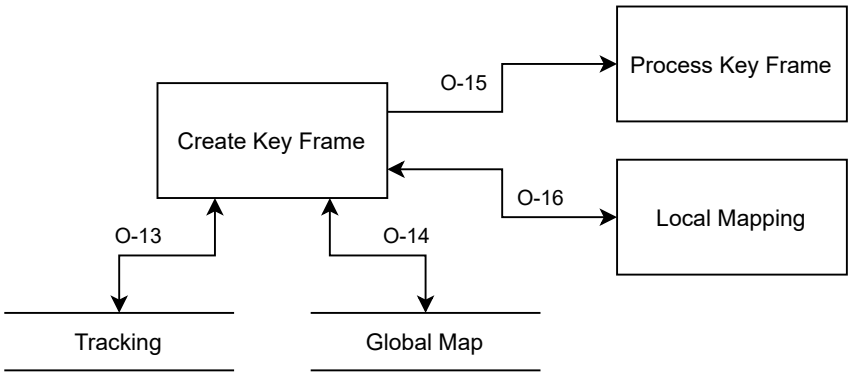


Figure 2.16: Exchange of information and interaction between the Create Key Frame functionality of ORB-SLAM2 and other parts of the system.

Table 2.16: Data dependencies for the Create Key Frame step in ORB-SLAM2.

Dependency	Direction	Description
O-13	In	Data related to the current Frame for decision making and creation of the new Key Frame.
O-13	Out	Storing references to the new (reference) Key Frame and Map Points.
O-14	In	Data related to the reference Key Frame and existing Map Points for decision making.
O-14	Out	Storing newly created Map Points in the Global Map.
O-15	Out	Handing off the newly created Key Frame to the Local Mapping thread.
O-16	In	Flags: loop closure pending, Local Mapping idle. Number: pending Key Frames.
O-16	Out	Flags: abort bundle adjustment, stop Local Mapping, a new Key Frame has been created.

Process Key Frame

For the sake of clarity, the first step of the Local Mapping thread is discussed separately from the other Local Mapping steps. This is done to create a better understanding of how responsibilities are divided between the Tracking and Local Mapping threads in ORB-SLAM2.

Process Key Frame takes each newly created Key Frame from the Tracking thread and inserts it into the Global Map. In order to prepare the Key Frame and associated Map Points for further use, all Map Point observations are now registered. Once all observations are correctly registered, the Co-visibility Graph information also has to be updated. If the original Frame of which the Key Frame was created was not processed by Track reference Key Frame or Relocalisation, the bag-of-words is also derived at this point.

Figure 2.17 shows the interaction of this step with other parts of the system. Process Key Frame also notifies the Local Mapping step of any Map Points that are new ("recently added") to the Global Map due to the creation of this Key Frame. These are the Map Points that were previously created during the Create Key Frame step. Table 2.17 summarises the information that is exchanged across each connection.

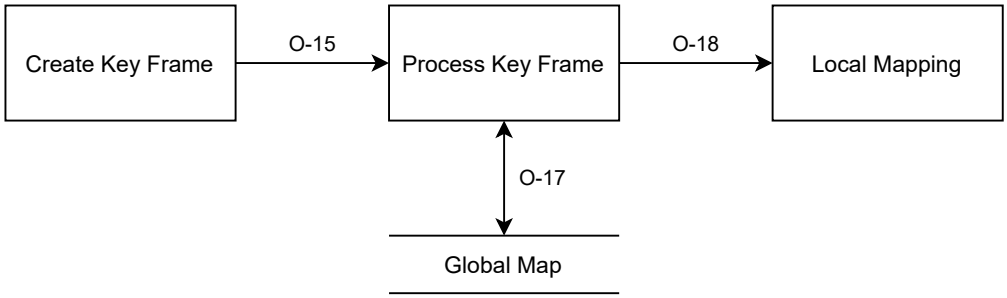


Figure 2.17: Exchange of information and interaction between the Process Key Frame functionality of ORB-SLAM2 and other parts of the system.

Table 2.17: Data dependencies for the Process Key Frame step in ORB-SLAM2.

Dependency	Direction	Description
O-15	In	Data related to the newly created Key Frame.
O-17	In	Data related to existing Key Frames and Map Points used for updating the Co-visibility Graph information.
O-17	Out	Storing the new Key Frame and registering observation Map Points.
O-18	Out	Recently added Map Points.

Local Mapping

The addition of a new Key Frame to the Global Map always triggers further processing by what is called Local Mapping. Local Mapping consists of a number of steps that all operate on a local sub-set of the Global Map. These steps are concerned with landmark detection, optimisation of (local) Key Frame and Map Point poses and removal of low quality Key Frames and Map Points.

The execution of the five steps listed below depends on the availability of Key Frames generated by the Tracking thread. The first two steps (Map Point culling and Create new Map Points) are always executed for every new Key Frame that enters the Global map via the previously

mentioned Process Key Frame step. Once these steps are finished, the third step (Search in neighbours) is only performed if no new Key Frames became available for processing. Similarly, the fourth and fifth steps (Local bundle adjustment and Key Frame culling) are only performed if there are still no new Key Frames created after Search in neighbours has finished. This approach shows that handling new Key Frames is prioritised over the Local Mapping operations, especially since the Create new Map Points and Local bundle adjustment steps can be interrupted and stopped by a new Key Frame becoming available.

- Map Point culling evaluates observations of Map Points that were recently added to the Global Map. This involves looking at the Found Ratio for the Map Point (based on observations by Frames in the Track Local Map step) and the total number of Key Frames observing the Map Point. Recently created Map Points are removed from the Global Map if not passing the required criteria after a short period of time. Map Points that passed this test remain in the Global Map until the number of Key Frames observing the Map Point drops to zero, due to other steps such as Key Frame culling and Local bundle adjustment.

- Create new Map Points detects new landmarks by matching features between the current Key Frame and its neighbour Key Frames. Neighbouring Key Frames are selected by using the Co-visibility Graph, in order to find those that already share the most observations with the current Key Frame. Only features that are not yet associated with the observation of a Map Point are then considered for matching. The matching process makes use of the bag-of-words of Key Frames to identify image features with a similar appearance. If a suitable match is found, the Map Point pose is determined by projecting the image feature coordinate into the world.

- Search in neighbours searches for more observations of Map Points that are already observed by the current Key Frame and the Key Frames in its neighbourhood. Compared to the previous step, no new Map Points are created and a larger set of neighbouring Key Frames is processed. Additional observations are found by projecting the existing Map Points into the camera coordinate frames of the Key Frames and matching against all features of these Key Frames. A successful match is registered as a new observation or replaces an existing observation if it is a better match.

- Local Bundle Adjustment optimises the estimated poses of local (co-visible) Key Frames and the Map Points they observe. This optimisation considers all observations for the bundle adjustment, such that the resulting pose estimates are based on the combined information carried by multiple observations of the same Map Point. Observations that are marked as outlier during this optimisation are removed from the associated Key Frames and Map Points in the Global Map.

- Key Frame culling removes Key Frames from the Global Map of which the majority of observations is not contributing to the quality of the observed landmarks.

After these tasks have been finished, or when a newly created Key Frame is available for processing, the current Key Frame is handed over to the Loop detection step. The information that is exchanged between the steps of Figure 2.18 is summarised in Table 2.18. Dependencies 0-16, O-20 and O-21 are only concerned with orchestrating the control flow between the Create Key Frame step (Tracking thread) and the Loop detection and Loop correction steps (Loop Closing thread). The other steps require knowledge about the state of the Local Mapping thread and are allowed to interrupt or disable the Local Mapping functionality.

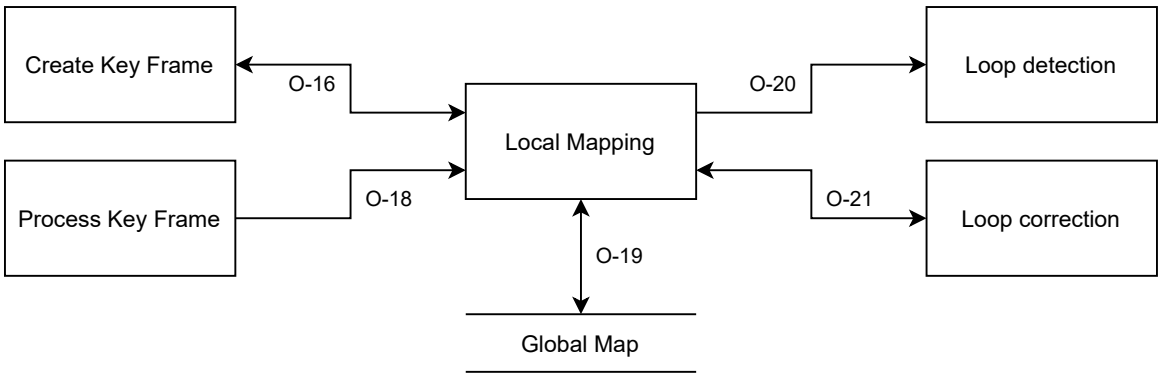


Figure 2.18: Exchange of information and interaction between the Local Mapping functionality of ORB-SLAM2 and other parts of the system.

Table 2.18: Data dependencies for the Local Mapping step in ORB-SLAM2.

Dependency	Direction	Description
O-16	In	Flags: abort bundle adjustment, stop Local Mapping, a new Key Frame has been created.
O-16	Out	Flags: loop closure pending, Local Mapping idle. Number: pending Key Frames.
O-18	In	Recently added Map Points.
O-19	In	Data related to the current and local Key Frames and associated Map Points. Used for landmark detection and matching, as well as pose optimisation via bundle adjustment.
O-19	Out	Storing updated observations and pose estimates for Key Frames and Map Points. Adding and deleting Key Frames and Map Points.
O-20	Out	Pass current Key Frame for processing by the Loop detection step.
O-21	In	Flags: abort bundle adjustment, stop/start Local Mapping.
O-21	Out	Flag: Local Mapping idle.

Loop detection

Each Key Frame that is created and added to the Global Map, is evaluated by the Loop Closing thread of ORB-SLAM2 after Local Mapping is done with it. The Loop detection step is first executed within this thread and is responsible for detecting the existence of loops in the robots trajectory. Like for the Re-localisation step, the bag-of-words of the current Key Frame is used to find Key Frames in the Global Map that observe features with a similar appearance.

Candidate Key Frames from this initial selection process, are further scrutinised by evaluating groups of close neighbour Key Frames and the landmarks they observe. Once these Covisibility Groups result in potential loop closure candidates, matched points between the current and candidate Key Frame are used to determine the transformation between the two ob-

servations (Horn (1987)). Further matching, optimisation and validation is performed, after which a loop closure is accepted if a sufficiently good match has been found. The resulting transformation describes the estimated translation, rotation and scale difference between the two Key Frames involved.

While the Co-visibility Groups cover a number of Key Frames and their observations, actual matching of image features is only performed between the current Key Frame and each individual candidate Key Frame. At this point, a final search for observed Map Points is performed by including the environment of the accepted candidate Key Frame. This last step is both validating the loop closure and providing additional matches for later use during the Loop correction step.

Figure 2.19 contains an overview of the interaction between Loop detection and other parts of the system. Details are listed in Table 2.19.

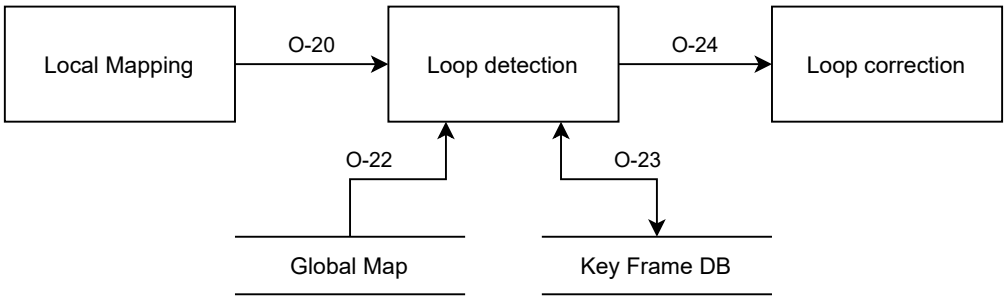


Figure 2.19: Exchange of information and interaction between the Loop detection functionality of ORB-SLAM2 and other parts of the system.

Table 2.19: Data dependencies for the Loop detection step in ORB-SLAM2.

Dependency	Direction	Description
O-20	In	Notifying the loop closure detection that a new Key Frame was added to the map.
O-22	In	Data related to all Key Frames and Map Points used for selection of loop closure candidates, matching features and estimation of the transformation describing the loop closure.
O-23	In	References to candidate Key Frames that potentially form a loop closure with the current Key Frame. Selection is based on the bag-of-words associated with each Key Frame.
O-23	Out	Storing (references to) the current Key Frame into the Key Frame database, to be used for future bag-of-words-based searches for candidate Key Frames.
O-24	Out	Matches and estimated transformation associated with the detected loop closure for further use during Loop correction.

Loop correction

Once an acceptable loop closure candidate is detected and the corresponding transformation between the current and candidate Key Frames is determined, the Loop correction step propagates this information to the Global Map. A selection of Key Frames and Map Points from the region around the current Key Frame is moved to the candidate Key Frame's location, by adjusting the associates poses using the transformation that describes the loop closure.

After this feed-forward correction, the regions surrounding the current and loop closure Key Frames are effectively laying on top of each other. As a consequence, different Map Points from both regions may actually represent the same physical landmark. These regions were separated before, but now the poses are corrected, the system is able to detect these duplicates. Likewise, Map Points that previously existed in only one of both regions, may now be used for matching against the other Key Frame's features to identify more observations of the same physical landmark. This warrants re-evaluation of the observation of Map Points by the current Key Frame, such that new matches can be found and duplicate Map Points can be discarded.

Once these operations that only cover a limited region are finished, the Loop correction step performs a pose graph optimisation. This optimisation covers all Key Frames in the Global Map and is called the "Essential Graph" optimisation. The Essential Graph is constructed from the ground up for every detected loop closure and contains the (feed-forward corrected) poses of all Key Frames.

Constraints/edges for the pose graph are generated according to several criteria, based on the connections in the Spanning Tree, the Co-visibility Graph and previously detected loop closures. The actual metric transformation that describes each constraint, is calculated from the Key Frame poses involved. Depending on the criterion that is met, the poses may be taken as they were before or after the feed-forward correction.

After the Essential Graph optimisation the estimated Key Frame poses are stored in the Global Map, replacing the previous estimates. The Map Points in the Global Map are also adjusted, by correcting their pose by the same amount as their associated reference Key Frames have been moved due to the optimisation.

Loop correction is finished by adding a "Loop Edge" to the two Key Frames involved, registering the occurrence of a loop closure for future reference. This is a bit of an inconvenient name in relation to pose graphs, as this edge does not describe a constraint between both Key Frames in any way. The Loop Edge only consists of a reference to the other Key Frame, such that the counterpart can be identified. The previously determined transformation that describes a metric constraint between the two involved Key Frames, is not stored or used any further after the feed-forward correction operation.

Figure 2.20 shows that this step mostly depends on information that is stored in the Global Map. Information about the existence of a loop closure is immediately provided by the Loop detection step. A summary of the information that is exchanged over these interfaces is given in Table 2.20.

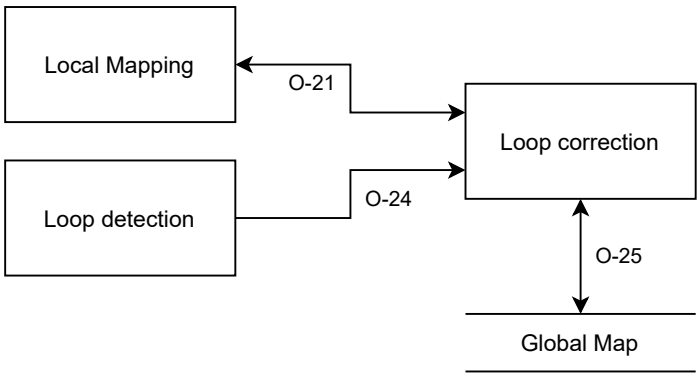


Figure 2.20: Exchange of information and interaction between the Loop correction functionality of ORB-SLAM2 and other parts of the system.

Table 2.20: Data dependencies for the Loop correction step in ORB-SLAM2.

Dependency	**Direction**	**Description**
O-21	In	Flag: Local Mapping idle.
O-21	Out	Flags: abort bundle adjustment, stop/start Local Mapping.
O-24	In	Matches and estimated transformation associated with the detected loop closure for further use during Loop correction.
O-25	In	Data related to all Key Frames and Map Points used for matching of observations and construction of the Essential Graph.
O-25	Out	Storing updated poses for Key Frames and Map Points, as well as registering updated observations and the loop closure itself.

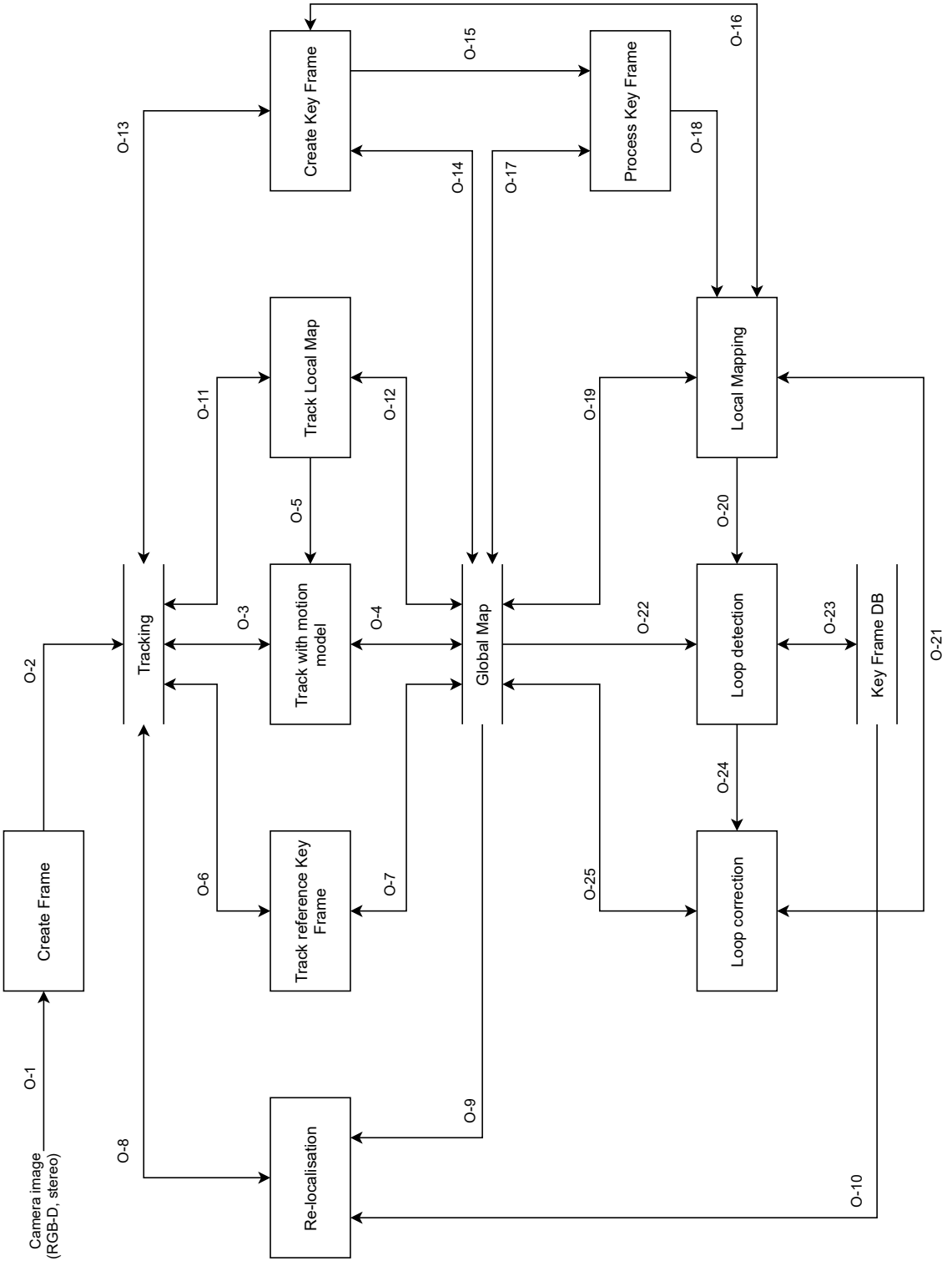


Figure 2.21: System overview of ORB-SLAM2 showing data dependencies between the identified functional steps.

2.3 Modularity for SLAM systems

Both RTAB-Map and ORB-SLAM2 do not seem to be designed and distributed with modularity of the algorithm itself in mind. Nonetheless, availability of the source code opens the door to reuse parts of these systems. The analysis in Section 2.2 provides insight into the functionality contained within these systems, with the associated dependencies indicating the exchange of information.

This information is used in this Section to discuss the architecture of RTAB-Map and ORB-SLAM2, considering the factors that aid or work against a modular approach for these algorithms. The discussion focuses on the dependencies between different parts of these systems and which responsibilities are fulfilled by each functional step. Some potential solutions for architectural factors that work against modularity, are also discussed.

Figure 2.22 at the end of this Section, provides an example on how RTAB-Map could look like if the proposed architectural changes are applied.

2.3.1 RTAB-Map

In general, RTAB-Map's code base is no ideal candidate for being reused, due to the dependencies caused by the use of system-wide types and objects. A big part of the functionality is implemented in two large classes, which makes it harder to separate parts for reuse. The documentation at GitHub and the associated publications are assuming that RTAB-Map is used as a complete system, even when using it as a C++ library.

Some key issues are highlighted in the discussion that follows next.

Threading

RTAB-Map's design primarily follows a single threaded execution of the steps shown in Figure 2.10. At a number of occasions, operations are dispatched to some helper threads, which allow for parallel execution, but the lifetime of these threads is short and limited to the specific operation that is performed at that given moment in time. This approach narrows the scope where these helper threads may affect information within the system, preventing the need for complex synchronisation schemes. As such, the algorithm behaves as the fixed sequence of operations as listed in Table 2.2 and modularity is not negatively affected by any temporal dependencies due to asynchronous behaviour.

Separated (external) odometry

Another aspect of the architecture that is visible in Figure 2.10 is the separation of the odometry functionality through the external input interface. Evidently, this allows for implementing odometry as a stand-alone module which can be reused or interchanged easily. This is facilitated by the definition of a suitable, restrictive interface which carries the information using widely adopted concepts (pose and covariance).

Short-term memory

The Short-term memory of RTAB-Map is involved in many operations, as witnessed by the number of dependencies that are shown in Figure 2.10. Especially when functional steps interact with both the short-term and working memories, modularity and reuseability are negatively affected due to the added complexity of these duplicated interfaces. As a result, any derived systems need to cope with the concept of two separate storage pools and the associated logic.

However, most of the affected functionality does not rely on the actual short-term memory concept as a whole. For example, Rehearsal only requires access to the data of the two most recently created Signatures (current and previous). This means that dependency R-6 in Table

2.4 does not specifically rely on the concept of short-term memory and all Signatures stored within. The same holds for Create Signature (R-3, Table 2.3), the Bayes filter (R-10, Table 2.5), Retrieval (R-14, Table 2.6) and the Loop closure Link (R-19, Table 2.7) steps.

The pose graph optimisation step already treats short-term and working memory as one storage pool, by optimising the poses of all Signatures in both memories. Therefore, dependency R-22 in Table 2.8 is only required by virtue of the existence of a short-term memory. Similarly, dependency R-7 which represents moving the oldest Signature in short-term memory to working memory is only there to facilitate the separation.

In all of these cases, the existing functionality does not rely on the actual separation between the short-term and working memory concepts. Therefore, the overall architecture could be greatly simplified by merging both memories together into one storage pool. This aids modularity by reducing the complexity of the interfacing for different steps. Code complexity also reduces, as the majority of the interactions with short-term memory are only there to cope with and enforce the separation between both memories.

Of course, short-term memory exists for a reason and merging the short-term and working memories affects the functional steps that rely on the separation between the two. This is the case for the Bayes filter (R-10, Table 2.5) and Transfer (R-25, Table 2.9) steps. Both steps can be modified to support the use of a single, merged memory and benefit from the reduced interface complexity. By moving the responsibility for distinguishing between recently created and older Signatures into the steps themselves, they are no longer depending on external factors. As an added bones, the criteria that separate short-term and working memory for both steps become decoupled, which enables independent tuning.

Flags and signalling between steps

Another cause for many of the dependencies shown in Figure 2.10 are signals that carry flags or counting values, used to exchange information about status and the occurrence of events. This complicates the reuse of functionality from the different steps, due to the additional interfaces imposed by these signals. The affected interfaces are concerned with the number of created and deleted Signatures (R-5, R-9, R-18) and triggering pose graph optimisation after alterations to the contents of short-term and working memory have been made (R-4, R-8, R-17, R-21).

The dependencies for these scenarios can be greatly simplified if the Transfer and Pose graph optimisation steps would be able to extract this information from the storage pools by themselves. By monitoring the contents of the attached storage for relevant changes, each step should be able to derive the relevant information or triggers by itself. This way, both steps become responsible for their own triggers or tracking metrics like the number of added Signatures. Modularity and reusability is greatly improved by these modifications, as the dependencies to other functional steps are completely removed.

Storage-induced dependencies

Solving dependencies by storing additional information within Signatures may not always be a good solution. For example, the weights assigned to Signatures during the Rehearsal step are stored in memory. Doing so creates an implicit dependency via R-6 and R-26, between Rehearsal and Transfer. However, in contrast to the flags and signals mentioned above, these weights are intended to be used during future iterations of the algorithm.

Whenever the choice is made to store information for future reference, the consequences should be carefully considered. In case of RTAB-Map's Bayes filter approach for detecting loop closures, a virtual Signature is constructed which represents the most commonly observed visual words. This Signature is also stored in working memory, via dependency R-11. As a consequence, all steps that interact with the working memory, need to be aware of its existence

to be able to ignore this specific Signature. Depending on the intended goals and context, such a construction can also be considered a leaky abstraction, which is discussed next.

Leaky loop closure detection and memory management

The final issue that works against modularity of RTAB-Map's functionality involves the loop closure hypotheses (R-12 and R-13, Table 2.5) and the Signatures selected for immunisation (R-18, Table 2.6). These dependencies carry information that is generated as intermediate result of the algorithm that is implemented and is not implicitly available from the attached storage pools.

When this kind of information crosses the boundary of a module or component via an interface, it leaks information about the underlying implementation or algorithm and becomes a leaky abstraction. Especially when this information is not easily represented in a format that is applicable for general use, the resulting interfaces impose the need for associated logic on both sides of the dependency. However, if there is such a tight coupling between both sides, the question raises if both steps are actually fulfilling a part of the same task. If that is the case, the associated functionality may be eligible to be merged, which completely deletes the dependency.

In general, there are several solutions to solve the dependencies mentioned above:

- Merge both steps into one module or component, which removes the need for an interface associated to this dependency.
- Move the behaviour that is responsible for generating the information which causes the dependency.
- Make this information available via the storage pools.

For example, merging Retrieval and Transfer does away with the dependency between the two steps (R-18), as is the case when merging the Bayes filter and Loop closure Link steps (R-13). But this does not solve the coupling between the Bayes filter and Retrieval (R-12).

Moving the responsibility for immunisation from Retrieval to Transfer also does away with the associated dependency (R-18). Similarly, if more of the decision making process is moved from the Bayes filter to the Loop closure Link step, R-12 and R-13 can be simplified by merging those into one interface which only carries knowledge about the highest hypothesis.

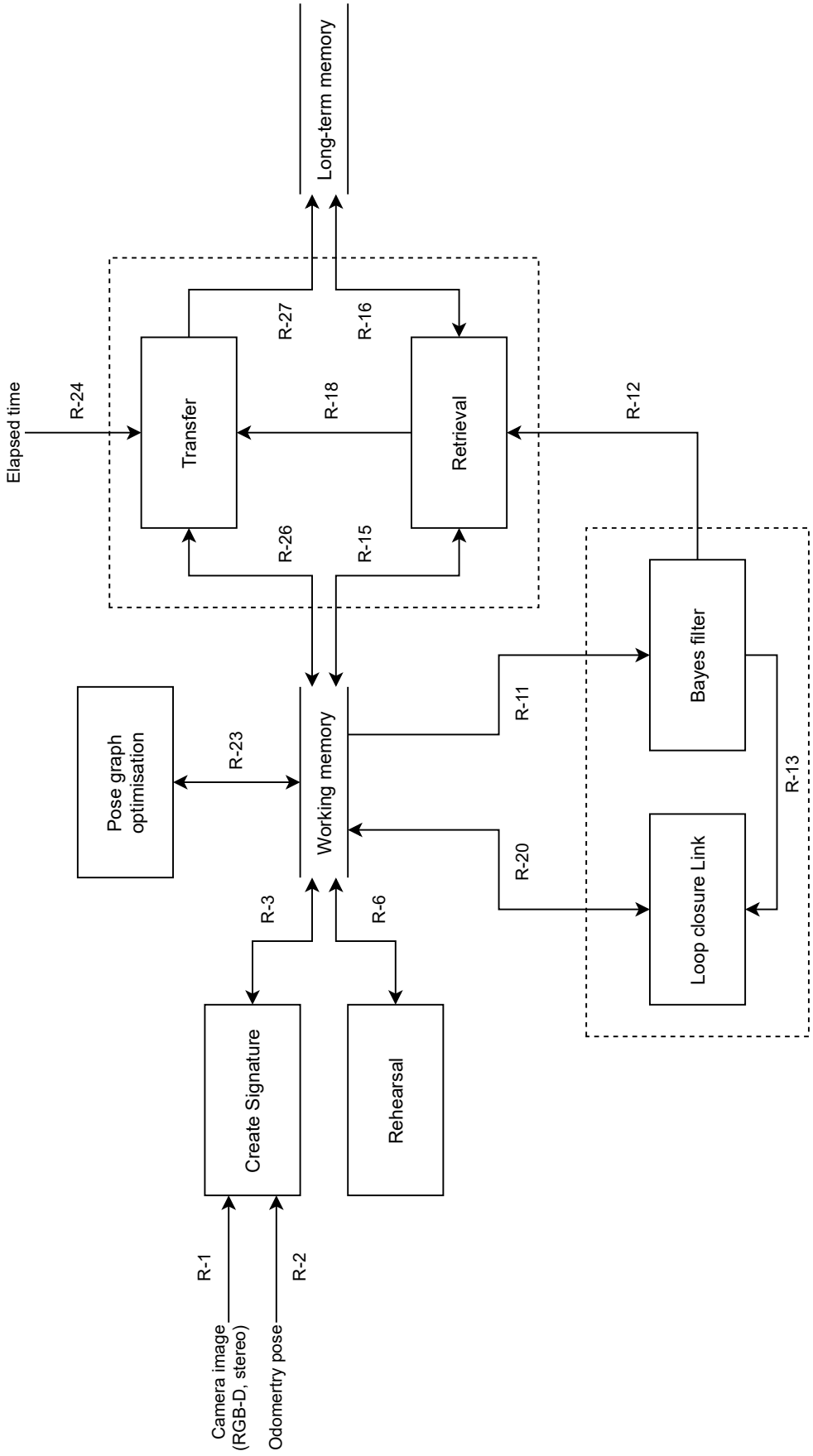


Figure 2.22: Example of an implementation for RTAB-Map with reduced complexity by applying the proposed architectural changes.

2.3.2 ORB-SLAM2

The steps that are executed by ORB-SLAM2 are mostly orchestrated by code that separates the control flow from the functions that are performed. This provides a convenient starting point when attempting to separate behaviour for reuse. Some parts may benefit from a better separation of responsibilities when designing a modular implementation.

The use of multiple threads enforces at least some modularity at an architectural level, but the vast amount of dependencies causes a tight coupling of behaviour between the different threads. This coupling occurs via information that is stored in the Global Map, which plays a central role in Figure 2.21. Direct dependencies between the functional steps are only sporadically used.

Separation between Tracking and Local Mapping

The use of Frames and Key Frames by ORB-SLAM2 defines an explicit boundary between two parts of the system. Unfortunately, this boundary is weakened by overlapping and weakly separated responsibilities between the Tracking thread context and Local Mapping thread context. Several mechanisms are causing this:

- The direct access to Key Frame and Map Point objects (which are fundamental to the Local Mapping thread) from within the Tracking thread context. Although it makes sense to use (read) the most recent estimates out of the Global Map, the current implementation is not using restrictive interfaces that minimise the scope of access by the Tracking thread functionality. Furthermore, these objects store data that is only relevant to the Tracking thread, data that is only relevant to the Local Mapping thread and data that is relevant to both. It is not clear which data is accessed when and modified by who. This makes reuse difficult because the effects of a modification are hard to oversee.
- Managing the "Found Ratio" directly from within the Tracking thread context. The Track Local Map step generates statistics about landmarks, required for the Local Mapping functionality.
- The creation and registration of landmarks (Map Points) in the Global Map, which happens within both threads. While Local Mapping should carry the main responsibility for this functionality, there seems to be a need for immediately registering newly detected landmarks from within the Tracking thread. The most likely cause for this is that these landmarks are to be used for the next iteration of the Tracking thread (the next camera image) and registration should occur before the Local Mapping thread would be able to do so.

A modular implementation would benefit from decoupling the functionality in the Tracking thread from concepts that are related to Key Frames, Map Points and the Global Map. This would be possible by implementing a more formal separation and mechanisms to sample information from the Global Map. However, these changes would affect functionality like the Found Ratio and requires a different approach to the creation of Map Points from within the Tracking thread.

Responsibilities in Tracking

There are also some issues that are specific to the Track Local Map step. A modular approach that does not required ORB-SLAM2's specific set of functionality may need to distinguish the three tasks that are fulfilled by Track Local Map:

1. Provide an updated estimate for the current pose. Only when this estimation step is successful, the motion model is updated.

2. Sample landmarks from the Global Map and match these against the current observation.

3. Determine the "Found Ratio" for observed landmarks during this step, which is required to remove low-quality Map Points from the Global Map that were recently created.

These tasks are also tightly coupled with the previous operations, by requiring an initial pose estimate and a set of matched landmarks to be provided. Again, a modular approach should define a suitable architecture to group this functionality in one or more components and formally define restrictive interfaces to exchange the relevant information.

Multi-threading and asynchronous access to data

Finally, a remark on the asynchronous nature of ORB-SLAM2. As seen from the Map Points and Key Frames in the Global Map, the information may change by the Tracking thread, Local Mapping thread, loop closure detection thread and Global bundle adjustment thread. Naturally, this requires some form of synchronisation mechanism to allow for atomic access and preventing the Global Map having an inconsistent state. Because the intention seems to be that each functional step should always use the most recent estimate, this means that every operation has to block all other operations. This makes an asynchronous parallel application functioning as if it was a regular, single threaded sequential program. But using asynchronous threads also comes at the cost of the need for all kinds of synchronisation and protection mechanisms, quickly adding up to all kind of workarounds to keep the data consistent. If not designed carefully, this works against reuse and modularity by cluttering the view on how the system works and causing all kind of errors due to race conditions and data inconsistencies.

3 Framework

The analysis of two real-world SLAM implementations in Chapter 2 provides insight into the functionality, architecture and dependencies within those systems. These insights will be used to construct a framework of modular components which facilitates improved reuse and adaptability for implementing SLAM algorithms. This effectively means rearranging the functionality that is covered by Figures 2.10 and 2.21 of the previous Chapter.

The first step towards such a framework for designing reusable components and SLAM systems is to develop a set of standardised components from which an algorithm can be constructed. These components are presenting a compromise between different topics presented in the previous chapters:

- Software engineering aspects and practices for modular and reusable code as listed in Section 1.4.
- Concepts and definitions from the SLAM theory presented in Section 2.1.
- Insights from the analysis of open source SLAM implementations in Chapter 2, covering functional behaviour contained within the systems (Section 2.2) and architectural issues that may affect modularity of the algorithm (Section 2.3).

Section 3.1 starts with defining the concepts and notation that is used to discuss SLAM algorithms in the context of the proposed modular framework. Using these definitions, Section 3.2 introduces a set of 6 component types, which become the building blocks for SLAM algorithms within the framework. At the end of this Section a basic example is given, to show how these components can be used to construct a SLAM algorithm. Section 3.3 then demonstrates how the functionality of RTAB-Map identified in Section 2.2 can be allocated to be implemented using the framework components.

3.1 Definitions and notation

This Section presents the concepts and notation that forms the base for the framework components and interfaces throughout the remainder of this Chapter. These definitions are a combination of the theory in Section 2.1 and insights from the SLAM systems in Section 2.2. The language associated with these concepts facilitates a consistent treatment of the workings of specific SLAM algorithm implementations and will be leveraged to derive modular components in the next Section.

Table 3.1 lists the definitions that are used throughout the remainder of this Chapter. The first part of the Table lists the inputs and outputs for a SLAM algorithm in general, which follows from the theory in Section 2.1. The second part list the entities that are distinguished within the information that is processed throughout a SLAM algorithm, and from which the SLAM outputs are constructed. A subscript t is used to identify the associated time index, while a colon is used to indicate a range: $i:j$. A superscript index or range indicates that there are more instances of the given type of entity at the given time index.

Some of the entries are associated with information fields, which applies hierarchy to the associated data. A map contains landmarks and a trajectory is constructed from poses. Samples, landmarks, correspondences and features carry values like the estimated pose x_t, as well as any other information such as features F, descriptors D and correspondences c_t.

The exact contents and format of these fields depends on the type of sensor and algorithm that is used. In case the features are extracted from a camera image, the associated coordinates on the image plane are commonly expressed as u and v. Alternatively, the definition in Section 2.1

also allows for a 3D-coordinate in the coordinate frame associated to the camera. Of course, these specific designators are different if the sensor type changes to one which measures range and bearing or time-of-flight.

Table 3.1: Concepts taken from SLAM theory to dissect and distinguish the different tasks that are performed by functional steps of the analysed SLAM algorithms.

Concept	**Notation**
Raw control	u_t
Raw measurement	z_t
Correspondences	$c_t = \{i, j, T, Q\}$
Trajectory or pose	$x_{1:t}$ or x_t
Map	$M = L_t^{1:n}$
Feature	$F_t^i = \{u, v, D\}$
Landmark	$L_t^i = \{c_t, x_t, D\}$
Sample	$S_t = \{u_t, z_t, c_t, x_t, F_t^{1:n}, ...\}$

- **Raw control u_t:** The conventional sensory input that carries information about the self-motion of the robotic platform for SLAM. Examples are the data from wheel encoders, motor control, accelerometers or gyroscopes.

- **Raw measurement z_t:** The conventional sensory input that carries information about the environment surrounding the robotic platform for SLAM. Examples are the data from optical/imaging sensors, radar, lidar, ultra-sonic sensors or force/touch-based sensors and switches.

- **Feature F_t^i:** A feature represents the lowest level entity which can be extracted from the measurement z_t. Features represent structures like points, lines or shapes that can be identified in the data from the measurement. Features can be associated with a coordinate in a reference frame attached to the sensor and descriptors D that specify properties of the detected feature. (This combined information forms the signature of a feature, as defined by Sebastian Thrun (2005).)

- **Landmark L_t^i:** As mentioned in Section 2.1, a landmark defines a distinct feature within the physical environment, associated to a specific position in the coordinate frame of the world or map. Like the features that can be extracted from the measurement, a landmark represents structures like points, lines or shapes. Along the associated pose, landmarks can contain other information like correspondences c_t and descriptors D, which can be used to identify distinct landmarks.

- **Sample S_t:** A sample represents all information that is available for a specific moment in time, which may involve the control u_t, the measurement z_t, detected features $F_t^{1:N}$, estimated pose x_t, correspondences c_t and any other fields that are describing the state of the robot and the environment at time t.

- **Correspondences c_t:** A correspondence establishes a (bilateral) relation between two of the entities mentioned above. As indicated in Section 2.1, establishing correspondence is often implicitly part of a SLAM algorithm. Correspondences are chosen to be explicitly included as they play an important role within the context of the presented framework. The types of correspondences and the information that is carried by them is discussed in more detail later in this Section.

- **Trajectory $x_{1:t}$ or pose x_t:** The trajectory contains a contiguous set of poses along the path followed by the robot. Each pose represents a coordinate in the world's coordinate frame and occurs with a specified number of degrees of freedom. All poses within a trajectory are expressed in the same reference frame. Different reference frames can be indicated by a superscript index: $x^i_{1:t}$.
- **Map M:** The (feature-based) map contains a set of landmarks and/or associated poses to represent knowledge about the structure of the environment. Like the poses of the robot in the trajectory, landmark poses are expressed as coordinates in the world's coordinate frame and occur with a specified number of degrees of freedom. All poses within a map are expressed in the same reference frame. Different reference frames can be indicated by a superscript index: M^i.

3.1.1 Types of correspondences

Correspondences throughout the SLAM algorithm can be established between the three types of entities listed in Table 3.1. To be able to distinguish between different operations and methods that establish correspondence, 6 possible types of correspondence are defined, which describe the relations that can be established between the entities mentioned above. Table 3.2 lists these 6 types of correspondences that can describe a relation between two of the previously mentioned entities.

Like samples, landmarks and features, correspondences can also be associated with additional information fields. These fields can carry values such as a measure of confidence/reliability, weights or a metric constraint. The example in Table 3.1 shows the notation for references to the involved entities i, j and a transformation with covariance between those entities T, Q.

When registering metric constraints with correspondences between two entities, the format which is used should be considered carefully. Although a correspondence is assumed to always describe a bilateral relation (no direction can be assigned to it), a constraint specified by a transformation or displacement vector does carry directional information. Each implementation should clearly specify how this directional information should be handled and interpreted, for it to able to be used in both directions along the correspondence. Also important when adding metric constraints to a correspondence is specifying the coordinate frame that is used.

Correspondences of a certain type can often be derived implicitly when correspondences of another type are known. This holds for example for deriving type-5 (landmark-sample) correspondence from type-2 (feature-landmark) correspondences between features associated to different samples. By knowing the type-2 correspondences between features and combining that information with knowledge about type-3 (feature-sample) correspondences between the features of a specific sample, the type-5 correspondence between the involved landmarks and sample can be established. The required type-3 correspondences are implicitly known whenever a sample comes with an associated set of features as part of its information fields. The same reasoning can be applied to the process of deriving type-6 (sample-sample) correspondence from type-1 (feature-feature) correspondences, if the associated samples with their features are available.

In any case, defining these relations explicitly allows for a better understanding of the information that flows along different interfaces. The proposed distinction between correspondence types provides the means to do this. Correspondence types can be specified by using the following extended notation:

$$c_t = C\#^i_t$$

Where # is used to indicate the correspondence type as listed in Table 3.2 and the indices i and t follow the conventions as introduced at the beginning of this Section. For example, when

two sets with j type-2 and k type-6 correspondences are to be indicated at the current time, notation becomes:

$$c_t = \{C2_t^{1:j}, C6_t^{1:k}\}$$

Table 3.2: Types of correspondences that can be determined between features, landmarks and samples.

Correspondence type	Description
C1. Feature - Feature	Establishes correspondence between two features.
C2. Feature - Landmark	Establishes correspondence between a feature and a landmark.
C3. Feature - Sample	Establishes correspondence between a feature and a sample.
C4. Landmark - Landmark	Establishes correspondence between two landmarks.
C5. Landmark - Sample	Establishes correspondence between a landmark and a sample.
C6. Sample - Sample	Establishes correspondence between two samples.

3.2 Framework components

This Section introduces the generalised set of components, each of which performs a task with a specific character within the framework.

The concepts listed in the previous Section are leveraged to identify common behaviour in the analysed SLAM systems. By using these concepts, further investigation of the functional steps of the previous Chapter yields insight into the information that is consumed or generated. These insights, together with the SLAM theory (Section 2.1) and considerations for modularity (Section 1.4 and 2.3), lay the groundwork for the component definitions that are presented next.

The first goal for the framework components is to separate responsibilities in a consistent manner and facilitate interchangeability. This is achieved by emphasising the type of output that is provided by a component. By fixing the output format for a specific component type, changing components becomes easy if changes are made to the available sensors or if there is a need to use different algorithms. Inputs to a component are allowed to be open for choice by the developer of the component. However, careful consideration of the required inputs for an algorithm is important, as these inputs may pose a burden on the applicability of a component for reuse in different systems. Inputs need to be kept restricted to only the information that is essential for the algorithm that is contained within the component. Preferably, these inputs make use of the generic concepts listed in the previous Section.

By looking at the definitions given in Table 3.1, the pose x_t, features F_t^i and correspondences c_t can be identified as primary elements. These three elements are derived from the inputs to the SLAM system and are the building blocks for the other concepts:

- Poses can be assigned to Samples and Landmarks.
- Features are associated with Samples and can represent the observation of a Landmark.
- Correspondences describe relations between the different entities.

Within the framework, three component types are defined, each of which is responsible for the generation of one of these primary elements. This results in the X-type, F-type and C-type components respectively.

For the estimation of the trajectory $x_{1:t}$ and map M, the O(ptimisation) and E(stimation) component type is defined. While the X-type component facilitates in the need for pose estimation for Samples and Landmarks, the responsibility for generation of the trajectory and map estimate is deemed a fundamentally different operation. This distinction results in an OE-type component which estimates the (full-SLAM) trajectory and/or map with landmark positions. The choice for a separate OE-type along the previously defined X-type component is inspired by the occurrence of reusable behaviour in the analysed SLAM systems. Thus, a separated X-type component improves the modularity for these kinds of operations.

The remaining components for the framework are responsible for two tasks that play an important supporting role within the analysed SLAM systems:

- Managing the data set on which the algorithm operates.
- Storing data for future reference.

These operations are supported within the framework by the D(ata) M(anagement) and D(ata) S(torage) component types respectively. Both components are performing a fundamentally different kind of operation, as the DM-type component is allowed to interact with the information provided via its inputs, while the DS-type component is only providing storage. All functionality contained within the DS-type component is only for the purpose of storing information and to allow access to the stored data. However, by storing the aggregated information produced by the other components, the storage component becomes responsible for constructing the actual Sample and Landmark entities.

Table 3.3 lists the 6 types of framework components of which a SLAM system can be constructed. In the remainder of this Section, each of the component types is discussed in more detail. This discussion highlights the considerations for the inputs and outputs for these components and provides context on how the assign behaviour to a suitable component type. At the end of this Section, an example is given on how these components can be connected together to compose a basic SLAM algorithm.

Table 3.3: Definition of component types to be used in the proposed modular framework. The first five components are performing operations which create or modify data. The sixth component does not contain functional behaviour other than what is required to fulfil the storage task.

Component type	**Description**
X-type	Generates an estimated pose x_t, associated to a Sample or Landmark.
F-type	Extracts features F_t^i and the corresponding descriptors D.
C-type	Establishes correspondences $C\#_t^i$ between entities as listed in 3.2.
OE-type	Optimisation and estimation, generates an estimate for the trajectory $x_{1:t}$ and/or map M.
DM-type	Data management, affects the (size of the) data set on which the calculations for the algorithm are performed.
DS-type	Data storage, stores information generated by the SLAM algorithm for later use, such as samples, landmarks and their associated information fields.

3.2.1 X-type components

The pose type component is defined to fulfil the need for pose estimation for both the robot and any detected landmarks. From the analysed systems became clear that many differ-

ent methods were repeatedly used to provide an updated pose estimate for the current robot location. RTAB-Map already facilitates a separated, external odometry implementation, while ORB-SLAM2 implements several pose estimation steps in the Tracking thread.

By requiring a fixed pose-type output, but allowing a free choice of inputs, different X-type component can be leveraged to adapt to the availability of different sensor types that contribute to the control u_t. In the absence of control data, the free choice of the input format also allows for the implementation of visual odometry methods, which are fundamentally based on the measurement z_t or derived features F_t^i. Likewise, the methods for the estimation of the pose of a landmark in visual SLAM are more similar to visual odometry.

3.2.2 F-type components

The feature type component is defined as a component which implements a method to produce or extract features from its input data. This component fits well with feature-based visual SLAM systems, as shown by the occurrence of feature extraction in both of the analysed systems. However, features can occur in a wide range of variants and types, which makes an interesting case for a separated component type that facilitates interchanging different implementations. F-type components can be easily swapped by using the fixed format F_t^i for their outputs, irrespective of the exact properties and information fields of these features.

Fixing the output format this way is only possible if the other parts of the system are able to process and use features in a way that is agnostic to the type and associated information fields. This is facilitated by using the two following mechanisms:

- Features contain standardised information fields, which describe common properties of features, like the coordinate and orientation.
- An F-type component is always accompanied by a matching C-type component, which is able to interpret the type-specific descriptors for matching and comparison.

Framework components that use features in their calculations, can directly use the values from the common property fields, while type-specific comparisons can be made using the appropriate C-type component.

In general, the input data for an F-type component is taken from the raw measurement data z_t. But different inputs are allowed to facilitate more elaborate methods for feature extraction, for example an algorithm that produces line- or surface-features from a set of point-features.

3.2.3 C-type components

Correspondence type components perform operations that are often encountered in literature as data association. Table 3.2 in the previous Section defined different types of correspondence, such that the character of different operations can be made more explicit. The output of C-type components is fixed by specifying the correspondence type for each output interface.

Like for the F- and X-type components, the inputs are free to be chosen by the developer, as this allows for an unlimited range of algorithms to establish correspondence. Preferably, the inputs are based on the previously introduced concepts, such as features F_t^i, poses x_t, landmarks L_t^i, samples S_t, etc. But correspondences may even be directly established from the information carried in the measurement z_t and control u_t.

Type-1 correspondences take a special place within the framework, as they are closely related to the associated feature types. C1-type components implement the matching and comparison functionality between features, to form a pair with the corresponding F-type component. The C1-type component contains knowledge about the feature-specific or descriptor-specific properties that are required for comparison. The resulting C1-type correspondences between

the affected features may carry additional fields describing a metric relation or a measure of similarity. Because these values can be expressed in generalised formats, the correspondences can be used by other components in the system without requiring further knowledge of the feature type that was used.

3.2.4 OE-type components

The optimisation and estimation type component contains the functionality which transforms stored information into an updated estimate for the poses in the trajectory and map. Evidently, the output is set to be a trajectory $x_{1:t}$ and/or a map M with the estimated positions of the landmarks. The input is allowed to be varied to be tailored to the implemented approach, but should at least specify a set of poses for samples or landmarks to be optimised and included in the trajectory and/or map. And of course, the required information or constraints that are required for the criterion which is optimised.

This approach allows for the optimisation to be interchanged with any other optimisation techniques, as long as the required data is available within the storage pools. By defining the boundary for the OE-components and requiring clearly specified inputs and outputs, it becomes easier to identify which data set is optimised and where to look for the results. Furthermore, this separation of functionality also fits well with the pose graph optimisation approaches that were identified in Section 2.2.

3.2.5 DM-type components

The data management component finds its origin in the practical application of SLAM algorithms in large environments. But rather than to detect or derive new information from its input data, its responsibility is to filter and manage the data set itself, in every way possible. Depending on the intended operation, a DM-type component can be placed "in-line" to operate on streaming data or it can be placed between storage pools to shuffle data around.

A significant part of the functionality in the analysed systems is responsible for managing the data set, such that the load by computational intensive operations is kept within reasonable boundaries. On top of this, the memory consumed by the data set cannot be left to grow without bounds, as computational platforms have a limited amount of RAM and non-volatile storage. These operations can be allocated to a dedicated DM-type component, which can be removed, replaced or reused when desired.

3.2.6 DS-type components

The data storage type component has a different character as the other components, as it does not implement functionality other than for providing storage. It serves as a source and destination for the other components within the framework. This component formalises storage as an explicit, separate responsibility.

Data storage as a tool helps to distinguish between data that is intended to be used in the future by other parts of the system and data that is of a temporary and local nature. This should improve reusability, as it defines a distinct source from which information can be retrieved.

DS-type components can implement storage for one or more of the following pieces of information:

- Samples S_t and Landmarks L_t^i and their associated information fields.
- C4-, C5- and C6-type correspondences ($C4_t^i$, ($C5_t^i$ and $C6_t^i$) and their associated information fields. These describe relations between the Samples and Landmarks and need to be stored at the same level as the entities they relate to.

- A single set of information fields to describe the status, properties and statistics for the storage component itself.

DS-type components provide separate interfaces to store and retrieve the different entities and correspondences. These input and output interfaces are allowed to be implemented in different ways, such as to allow per-field or per-entity access (or even all stored data at once).

Any DS-type implementation is free to choose which entities it supports to be stored. Likewise, the framework does not specify any required fields for these entities. It is left to the algorithm developer to identify the information that should be stored for future reference. As a result, DS-type components can be targeted at maximising flexibility by supporting storage for any information field that is offered at its interfaces, or targeted at robustness and reduced overhead by defining a fixed structure for the information fields.

3.2.7 Integration example

Figure 3.1 contains an example on how to use the components presented in this Section for the construction of a SLAM algorithm. In line with the subject of this work, the demonstrated algorithm implements a basic feature-based, visual, pose graph SLAM system with loop closure detection. The inputs u_t and z_t consist of wheel encoder values and camera images, of which the format is not further specified for this example.

This algorithm implements the following behaviour:

- Image feature extraction of an unspecified kind, such as ORB, SURF, SIFT, etc. This is implemented in the associated F- and C-type components in Figure 3.1.
- Wheel encoder based odometry, implemented in the X-type component which uses the encoder values in u_t to estimate the current pose x_t.
- Landmark detection by comparison of current and past image features (C1-type correspondence), according to an unspecified algorithm. The associated C-type component implements this behaviour, while the landmark detection X-type component provides a pose estimate for each detected landmark.
- Loop closure detection by comparison of landmarks observed by the current and previous Samples (C5-type correspondence), according to an unspecified algorithm. The loop closure detection is implemented in a dedicated C-type component.
- Pose graph optimisation which estimates the poses for both the trajectory $x_{1:t}$ and landmarks in the map M. The associated OE-type component is clearly visible in Figure 3.1.
- A single storage pool is used, indicated by the associated DS-type component.

The annotation at the interfaces between the components in Figure 3.1 indicate the information that is exchanged and follows the notation introduced in Section 3.1. Some interesting observations about this implementation follow directly from the Figure and the presented structure:

- The inputs u_t and z_t are consumed by the odometry and feature extraction components respectively. Both components contribute to the current sample S_t, by feeding their output to the storage component. As a result, samples in the storage component contain the corresponding information fields: $S_t = x_t, F_t^i$.
- The F-type components is accompanied by a C-type component which performs the feature-specific matching and comparison operations. Feature matching (C1-type correspondence) is required for the landmark detection and loop closure detection C-type

components. The Figure shows the corresponding matching interfaces to and from the feature matching C-type component. As a consequence, both components require features at their inputs, which are provided with the Samples from storage.

- Additional to the Samples that are entering storage, the DS-type component also contains Landmarks with poses, C5-type correspondences between Samples and Landmarks and C6-type correspondences between Samples. This is immediately visible by the corresponding interfaces that feed data to the storage from the landmark detection X-type, landmark detection C-type and loop closure detection C-type components. The C5-type correspondences from storage are used by the loop closure detection and pose graph optimisation components, while the C6-type correspondences are only used for pose graph optimisation.
- Both the landmark detection and the loop closure detection operate on the same set of Samples. Evidently, the loop closure detection also requires C5-type correspondences for its algorithm.
- The pose graph optimisation estimates the poses of the trajectory $x_{1:t}$ and landmark poses for M based on the odometry pose of the stored samples $S_{1:t}\{x\}$ and the estimated pose for the landmarks $L_{1:i}\{x\}$. The C5- and C6-type correspondences from storage are required to generated the constraints or edges for the pose graph. The Figure also shows that the resulting optimised poses in the trajectory and map are never written back to storage for future use.
- Data management is not performed by this algorithm, indicated by the absence of a DM-type component.

One important aspect of the components is not visible in Figure 3.1. The required information fields for a specific component are not indicated by the S_i, L_i and $C\#_i$ annotation of the corresponding input and output interfaces. For this specific example it would be possible to add the required information fields to the annotation of each interface, due to its simplicity. However, the view will become cluttered very rapidly for more complex systems that use a higher number of different information fields for Samples and Landmarks. In general, Figures should only specify the types of entities and correspondences that are exchanged between components, while the accompanying documentation should specify the specific information fields.

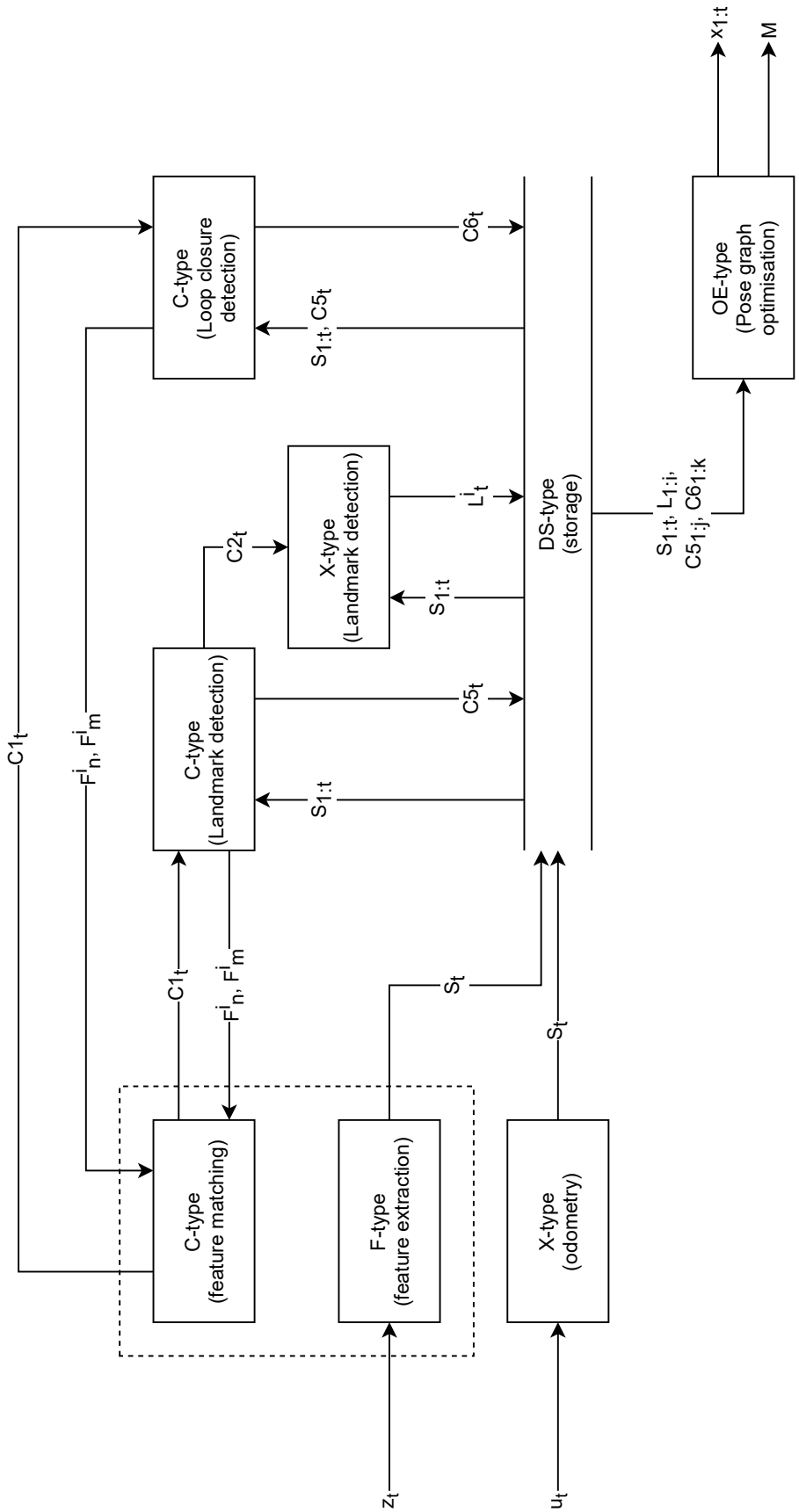


Figure 3.1: Example of a generic feature-based, visual, pose graph SLAM algorithm when implemented using the proposed framework components.

3.3 RTAB-Map

Using the definitions presented in Section 3.1 and the component definitions from Section 3.2, the functional steps identified in Table 2.2 can be translated to a modular implementation. When done right, the issues highlighted in Section 2.3 can be solved, such that reusability and interchangeability improves.

With Figure 2.22 as a starting point, some additional rearrangement of behaviour is required to fit the framework components. Table 3.4 lists the resulting framework components to which the original functionality is mapped. Each new component is given an appropriate name for identification, while the Table also lists the functional step(s) from which the behaviour originates.

For each component, a short description follows, which explains the main considerations for arranging the behaviour of RTAB-Map this way. The resulting input and output interfaces for each component are also discussed.

Table 3.4: Framework components to which the original RTAB-Map functionality is assigned in order to implement the algorithm in a modular fashion.

	Component	**Type**	**Original functional step(s)**
1.	Feature extraction	F-type	Create Signature.
2.	Feature matching	C-type	Rehearsal, Bayes filter and loop closure Link.
3.	Odometry	X-type	-
4.	Loop closure detection	C-type	Bayes filter and loop closure Link.
5.	Rehearsal	DM-type	Rehearsal.
6.	Memory management	DM-type	Retrieval and Transfer.
7.	Pose graph optimisation	OE-type	Pose graph optimisation.
8.	Working memory	DS-type	-
9.	Long-term memory	DS-type	-

3.3.1 Feature extraction and matching: F- and C-type

The extraction of features and descriptors during the Create Signature step is allocated to a single F-type component. RTAB-Map offers a range of choices for the type of features and descriptors that are used by the system. The same range can be offered within the proposed framework, by implementing a single component with support for all of these types, or by implementing separate F-type components for each specific type. In any case, once RTAB-Map is running the feature type is fixed. Therefore, the remainder of this work treats the feature extraction in RTAB-Map as a single F-type component, without further specifying the feature type.

The purpose and properties of visual words for the bag-of-words approach are very similar to the conventional descriptors for image features. During the original Create Signature step, the bag-of-words for a Signature is derived following the extraction of image features and descriptors. Therefore, this responsibility is also added to the new F-type component for feature extraction.

Comparison of RTAB-Map's image features can be achieved by using the conventional descriptors, or by using the visual words from the bag-of-words approach. The C-type component which performs these operations for RTAB-Map facilitates both methods. For the conventional descriptors, this means that the appropriate OpenCV matching routines are used, while the visual words are matched by simply comparing the integer values of their Ids.

Of course, the purpose of using a bag-of-words approach is to quickly search for samples that observe features described by the same visual words. This functionality is also allocated to this feature-specific C-type component, as it relies on the visual word descriptors. The C-type component also maintains the associated (dynamic) dictionary functionality and inverse references to samples, which are required for the comparison and matching tasks.

The pair of F- and C-type components are shown in Figure 3.2. A specification for the corresponding interfaces is given below:

- The input z_t for feature extraction contains a stereo or RGB-D camera image.
- The output F_t^i of the feature extraction consists of i features. These features contain fields for 2D image plane coordinates ($\{u, v\}$), 3D camera frame coordinates ($\{x, y, z\}$), binary feature descriptors (D_f) and visual word descriptors (D_{vw}):
  - $F_t^i = \{\{u, v\}, \{x, y, z\}, D_f, D_{vw}\}$
- Both feature matching inputs require two descriptors of the same type, the conventional binary descriptors D_f, or the visual words D_{vw}. Naturally, the matching input for lookup of samples by their observed visual words only supports the visual word type descriptor.
  - $\{F^i = D_f, F^j = D_f\}$
  - $\{F^i = D_{vw}, F^j = D_{vw}\}$
  - $F^i = D_{vw}$
- The outputs for the feature matching consist of C1- and C3-type correspondences. For the feature matching by binary descriptor or visual words, the C1-correspondence contains fields to refer to both involved features. Similarly, the inverse lookup provides a C3-correspondence with fields to refer to the involved visual word descriptor and sample. The outputs return an empty set when no match was found.
  - $C1_t = \{F^i, F^j\}$
  - $C1_t = \{F^i, F^j\}$
  - $C3_t = \{F^i, S_j\}$
- The feature matching component implements dedicated interfaces to control both the contents of the (dynamic) visual word dictionary and the references for the inverse lookup of samples based on visual words. RTAB-Map's dictionary of visual words is dynamically constructed from the samples that reside in the short-term and working memories. No further specification for this interface is given, as this involves a detailed discussion of the desired control flow and the order of operations within RTAB-Map's algorithm. Depending on the specific needs, standardised formats can be used to offer the required descriptors or correspondences for the component to be able to add, remove and update the dictionary and references.

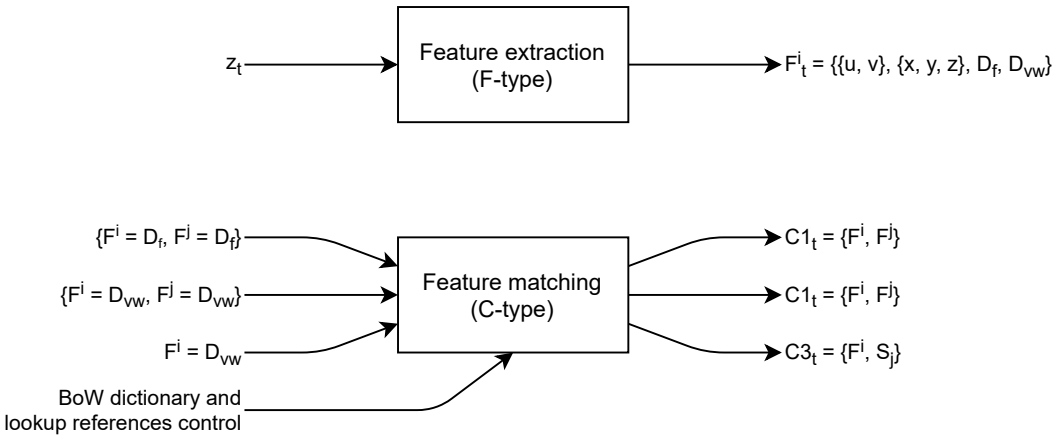


Figure 3.2: F- and C-type feature extraction and matching components for the implementation of RTAB-Map within the modular framework.

3.3.2 Odometry: X-type

Odometry within the core algorithm of RTAB-Map consists of a layer which passes the externally provided odometry information through to the system. The corresponding interfaces already implement some form of abstraction, which can be used as inspiration for the X-type component.

This X-type component implements the same abstraction layer within the framework implementation. Although the abstraction may not be specifically necessary in this case due to the identical input and output formats, defining this X-type component serves as an example.

Figure 3.3 shows the basic definition for an odometry component for RTAB-Map, which estimates a pose x_t from an unspecified sensory input u_t.

- The input u_t requires an odometry pose x_t, with associated covariance Q to indicate the uncertainty of the pose estimate. RTAB-Map expects a pose formatted as a transformation from the origin of the odometry reference frame to the current position, providing 6 degrees of freedom.
  - $u_t = \{x_t, Q\}$
- The provided input is immediately propagated to the output:
  - $x_t = \{x, Q\}$

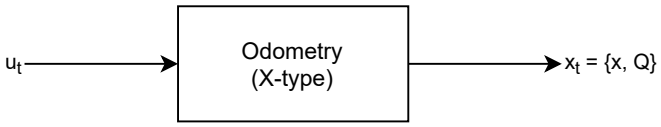


Figure 3.3: X-type odometry component for the implementation of RTAB-Map within the modular framework.

3.3.3 Loop closure detection: C-type

As suggested in Section 2.3, the functionality of the Bayes filter and Loop closure Link steps of RTAB-Map are merged into a single C-type component. Both original steps contribute in part to the decision making process for detecting loop closures, which makes the separation between the two inconvenient from a responsibility point of view. By merging both steps, all operations that are required for the loop closure detection are encapsulated into one component.

While the information that is originally part of dependency R-13 can now be kept internally, the information carried by R-12 and R-20 still needs to be available from the component output. This information consists of two parts:

- The "highest hypothesis", as determined by the Bayes filter. This describes a relation between the current sample S_t and a sample from working memory S_{t-i} that is associated with a previously made observation.
- The actual occurrence of a loop closure between the same samples that are associated to the "highest hypothesis", for which a metric constraint is determined by estimating the transformation between the two image coordinate frames.

There are several solutions for solving this problem, depending on how the responsibilities are separated between this and other components.

- This C-type component is responsible for the final decision on the existence of a loop closure: two separate C6-type outputs can be used to represent the highest hypothesis and loop closure concepts. Alternatively, a single C6-type output can be used. The availability or absence of an associated transformation in the correspondence can signal the existence of a loop closure. This approach is less favourable, as it requires knowledge about these concepts in the components that consume the output.
- A separate C-type component is responsible for the final decision on the existence of a loop closure: a single C6-type output can be used to represent the highest hypothesis of the Bayes filter. Effectively, this separates the loop closure detection again into two components like the original implementation. This approach allows for mimicking the original flow of execution of RTAB-Map, at the cost of complicating the reuse and interchangeability for the loop closure detection as a whole.

Lacking the evidence that the original Retrieval step is required to be executed before the Loop closure Link step[1], the single C-type component approach is chosen to reduce the complexity of the overall structure.

Furthermore, the "virtual Signature" sample is removed from storage, such that other external components do not have to take its existence into account. Instead, this sample can now be maintained as part of the internal state of the loop closure detection component.

This results in the loop closure detection component shown in Figure 3.4. The following input, output and matching interfaces are specified:

- The main input to the loop closure detection component are the reference features $F_t^{1:i}$ for the current sample S_t and the set of features $F_{j:k}^{1:i}$ associated to previously created samples $S_{j:k}$. For all of these features, descriptors are required to perform the matching and lookup parts for the loop closure detection. The type of the descriptors actually follows from the components that are attached to the matching interfaces. In RTAB-Map's implementation, both use descriptors of the visual word type. For the features of the current sample S_t only a 2D-coordinate is required by RTAB-Map, instead of a 3D-coordinate for the other features.
  - $S_t = F_t^{1:i} = \{\{u, v\}, D\}$
  - $S_{j:k} = F_{j:k}^{1:i} = \{\{x, y, z\}, D\}$
- Two separate C6-type outputs are generated, such that the system is able to distinguish between the "highest hypothesis" correspondence without having an associated trans-

[1] Retrieval affects the inverse lookup references for visual words, but this functionality is not used by the Loop closure Link step.

formation and the actual "loop closure" correspondence which does contain a transformation T as a constraint. The uncertainty of the transformation is also given by the covariance Q. Both correspondences contain the references to the involved samples S_t and S_n.

  - $C6_t = \{S_t, S_n\} | n \in j:k$
  - $C6_t = \{S_t, S_n, T, Q\} | n \in j:k$

- The remaining matching interfaces in Figure 3.4 are used to find correspondences from the visual words. These interfaces correspond to the inverse bag-of-words lookup to find samples that observe the same visual words (C3-type) and regular feature matching (C1-type) as previously defined.

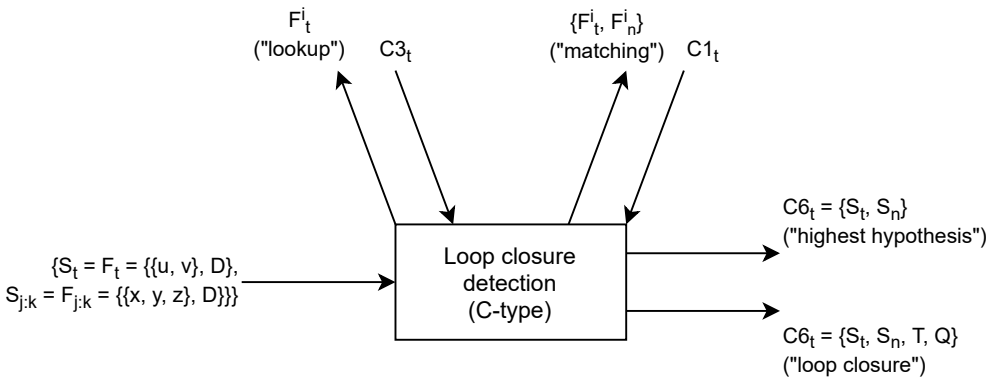


Figure 3.4: C-type loop closure detection component for the implementation of RTAB-Map within the modular framework.

3.3.4 Rehearsal and Memory management: DM-type

The Rehearsal step in RTAB-Map is primarily concerned with limiting the rate at which new samples enter the system for being processed. This makes Rehearsal a prime candidate for allocation to a DM-type component.

The Retrieval and Transfer steps of RTAB-Map are responsible for managing the separation between two distinct data sets with samples: working memory and long-term memory. Following up on the suggestions in Section 2.3, the Retrieval and Transfer steps of RTAB-Map are allocated within a single memory management DM-type component. This encapsulates the R-18 dependency between both steps, which is associated with the responsibility for immunising samples from the Transfer step. Other than adding a lot of complexity to the system, the short-term memory concept did not contribute significantly to the algorithm and is not carried over to the new memory management component. As a result, the memory management approach becomes easier to interchange or reuse.

There still exists a dependency between the proposed Rehearsal and Memory management components, as the weight values for samples are determined during the Rehearsal step. Memory management evaluates these values as part of the selection criteria for Transfer. This dependency is fulfilled in the original implementation by storing the weights as part of the Signatures. Because a weight as a concept is sufficiently generic, this can be carried over to the framework component by adding the weight as an information field for the samples. RTAB-Map's memory management can also function when weights are not available. Modularity and reusability is greatly improved if the weight is defined as an optional input field to the DM-type component.

Another dependency related to the memory management is due to the measurement of execution time of (a part of) the algorithm. Of course, the first consequence is that there should be a mechanism to provide the time measurement to the Memory management component. This can be solved by directly feeding the elapsed time to the component, by providing two time stamps for the beginning and end of the measurement, or by just signalling the beginning and end of the measurement window. Each of these approaches results in a different balance of responsibilities between the Memory management component and the surrounding system, causing different constraints on the reusability. For the remainder of this work, the measured execution time is assumed to be available and passed as a single value to the Memory management component.

The second consequence of this Memory management approach are an implicit requirement on all operations that are performed during the window of the execution time measurement. For the measurement to give an accurate representation of the computational load, execution time of the algorithms that are monitored should not vary significantly in relation to the code-path that is followed[2]. Ideally, the execution time should only depend on the size of the data set that is used. This puts a heavy requirement on reusability, as the software surrounding the Memory management component needs to take this into consideration. Generally, developers are used to do exactly the opposite by using early returns and preventing heavy computations if the initial checks fail. Intentionally causing a constant load may also work against the energy efficiency of the algorithm.

The resulting DM-type components for RTAB-Map are shown in Figure 3.5. Both components treat information on a per-sample base, being the current S_t and previous sample S_{t-1} for Rehearsal and those contained in working memory $S_{i:j}$ and long-term memory $S_{k:l}$ for Memory management.

The minimum requirements for the information fields for these samples are as follows:

- Both input samples for Rehearsal are required to contain fields for the pose x_t, weight w_t and features F_t^i with visual word descriptors D_{vw}:
  - $S_t = \{x, w, F^i\}$, with $F^i = D_{vw}$
  - $S_{t-1} = \{x, w, F^i\}$, with $F^i = D_{vw}$
- The output for Rehearsal consists of one or both samples from the input, depending on if a merge happened. The weights are adjusted accordingly:
  - $S_t = \{x, w, F^i\}$, with $F^i = D_{vw}$
  - $S_{t-1} = \{x, w, F^i\}$, with $F^i = D_{vw}$ (Absent in case of a Rehearsal merge)
- All samples for the inputs and outputs for the Memory management component are required to contain fields for the pose x_t, weight w_t and features F_t^i with visual word descriptors D_{vw}:
  - $S_{i:n} = \{x, w, F^i\}$, with $F^i = D_{vw}$

For the visual word matching interfaces, refer to the corresponding feature matching C-type component discussion above. The interfaces for the execution time measurement and dictionary management are not further specified at this time.

[2]This problem is closely related to the field of IT security, where time measurements are used as an side channel attack to distinguish between different code-paths that are executed. This information can be used to extract information from the executed algorithm. To mitigate these issues, the execution time of different code-paths needs to be made equally long.

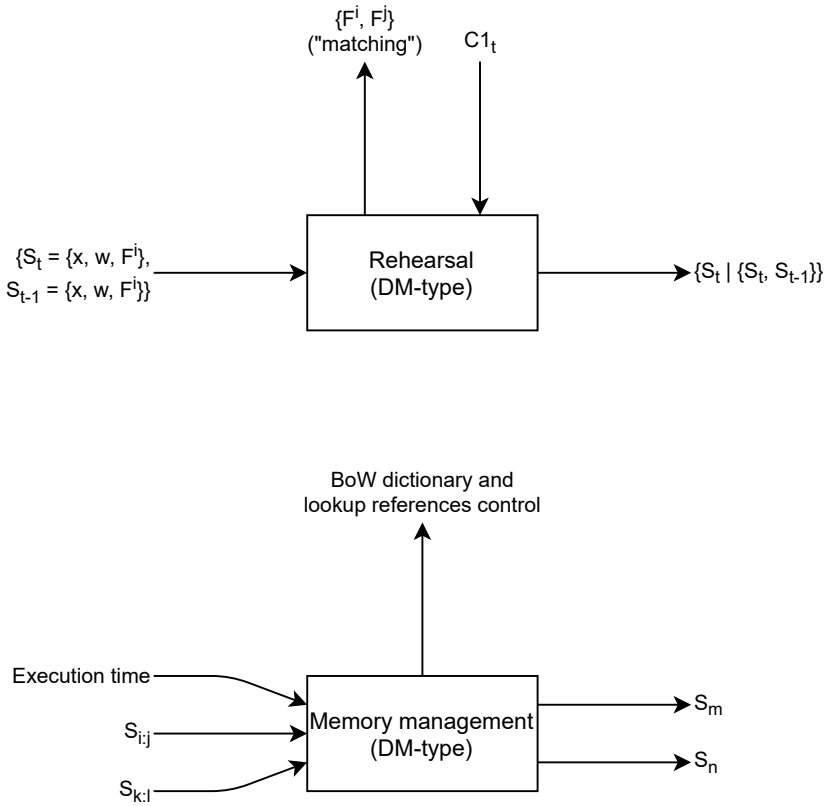


Figure 3.5: DM-type Rehearsal and Memory management components for the implementation of RTAB-Map within the modular framework.

3.3.5 Pose graph optimisation: OE-type

The pose graph optimisation in RTAB-Map is by itself already mostly a stand-alone component. However, the code that constructs the pose graph is tightly coupled to the memory management approach. Moving this functionality to the proposed OE-type component and using generalised interfaces to provide the relevant data set(-s), removes this coupling.

The poses and constraints that become the pose graph need to be derived from the available input data. From the analysis in Section 2.2 it is known that RTAB-Map does not process landmarks through the core of its algorithm. Therefore, the pose graph only contains poses of the samples that contribute to the trajectory $x_{1:t}$.

Figure 3.6 shows the required inputs and outputs for the resulting OE-type component. The samples $S_{i:j}$ and correspondences $C6_{k:l}$ at its inputs do require the following fields:

- For each sample that is represented by the pose graph, the associated pose x_t and covariance matrix Q are required. The input for C6-type correspondences (corresponding to RTAB-Map's Neighbour and Loop closure Links) requires fields i and j to identify the involved samples and carry a transformation T with covariance matrix Q to define the metric constraint:
  - $S_{i:j} = \{x, Q\}$
  - $C6_{k:l} = \{i, j, T, Q\}$
- The output of the pose graph optimisation only contains the optimised poses $x_{i:j}$:
  - $x_{i:j} = x$

RTAB-Map makes a distinction between the estimated poses that the result of the pose graph optimisation and the original odometry poses for each sample. For the functionality of RTAB-Map as discussed in this work, the optimised poses are only required as initial guess for the pose graph optimisation itself. Therefore, the optimised poses can be kept as internal state within the OE-type component, without the need for a dedicated "guess" input. This simplifies the interfaces to the pose graph optimisation component, which is beneficial for the reusability of the component.

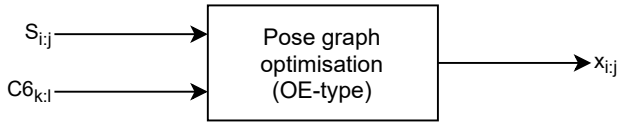


Figure 3.6: OE-type Pose graph optimisation component for the implementation of RTAB-Map within the modular framework.

3.3.6 Working and Long-term memory: DS-type

Following up on the suggestions made in Section 2.3, two storage pools will be used for the modular RTAB-Map approach. The character of these two pools is different, as the combined short-term and working memory pool keeps data in RAM, while the long-term memory implementation involves the use of an SQL database. This warrants the use of two separate DS-type components, providing the two different implementations.

As mentioned before, RTAB-Map makes a distinction between the odometry poses and optimised poses for the samples that are involved with the pose graph optimisation. The original implementation stores the odometry poses with the samples in the short-term, working and long-term memories, while separate variables in RAM are used to keep track of the optimised values for future reference. Although the responsibility for the associated storage was removed by incorporating an internally tracked state within the Pose graph optimisation component, this particular case serves as a good example for when availability for future reference has to be made explicit.

Following the original implementation, RTAB-Map keeps track of the odometry pose and optimised pose for Signatures in short-term and working memory. Within the framework, this can be made to be shown explicitly by defining two separate information fields for samples in storage: $S_i = \{x_{odometry}, x_{optimised}\}$. This shows that both values are stored in the associated DS-type component(-s) for future reference. Likewise, components that require these values from storage, need to add these fields to the specification of their input interfaces.

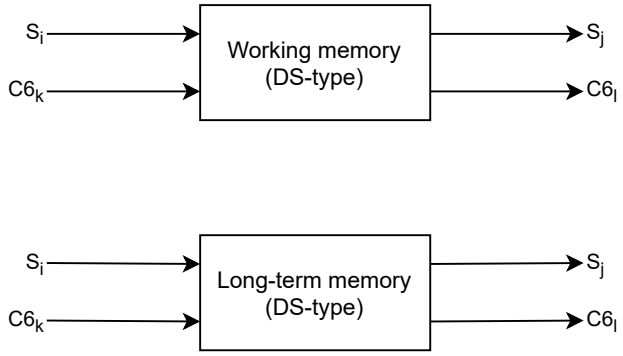


Figure 3.7: DS-type Working memory and Long-term memory components for the implementation of RTAB-Map within the modular framework.

3.3.7 Integration

The components listed in Table 3.4 cover all of the original behaviour from the steps listed in Table 2.2. Figure 3.8 shows how these components are combined to construct the SLAM algorithm. As these framework components attempt to solve some of the modularity issues discussed in Section 2.3, it is not surprising that the structure looks very similar to that of Figure 2.22.

As was the case for the example in Section 3.2, the Figure does not show information about the actual information fields that are required for the samples and correspondences. The previously treated descriptions for each component are providing the associated documentation about the required information fields.

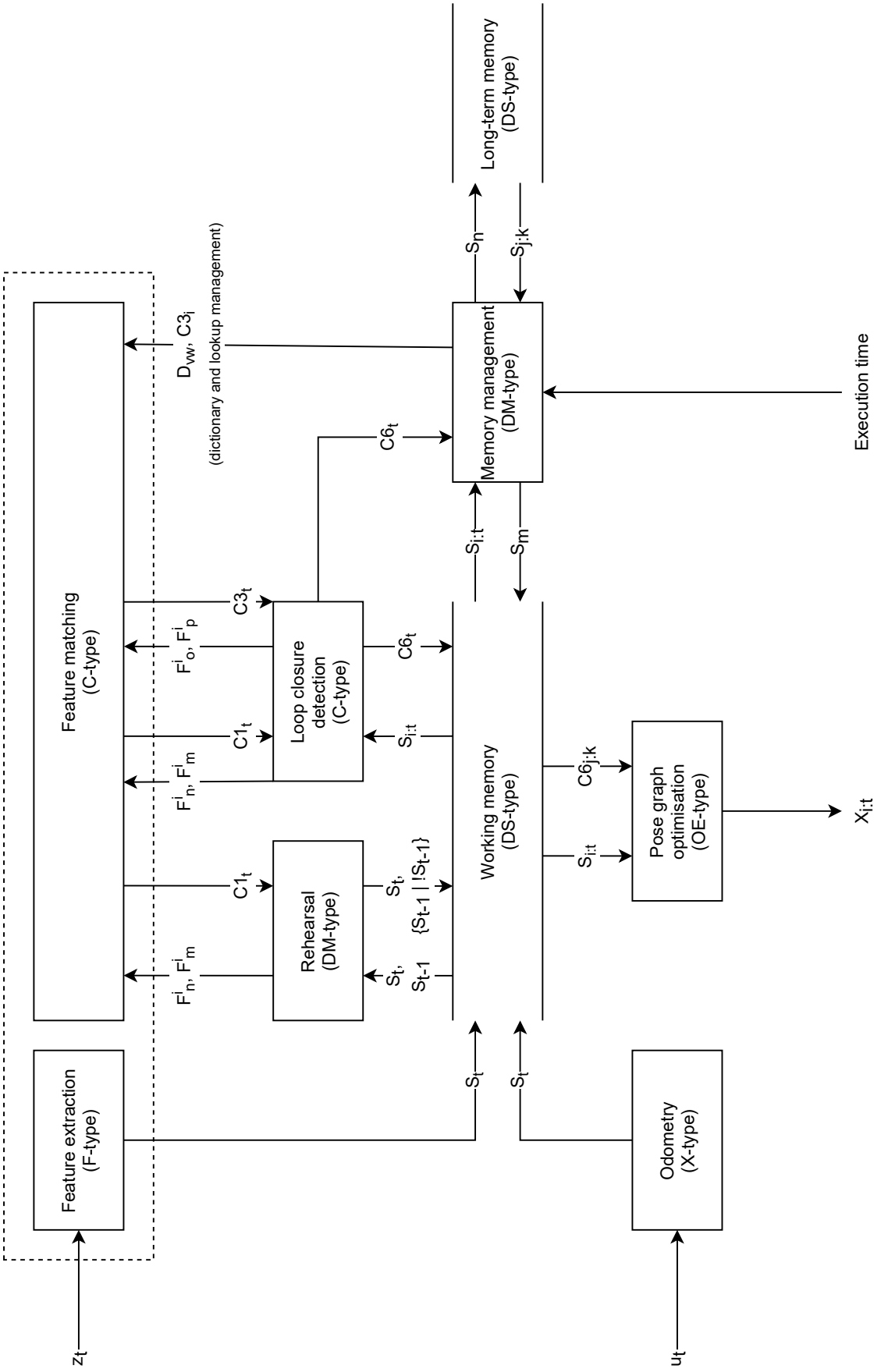


Figure 3.8: Implementation of RTAB-Map within the modular framework, made by composition of the previously derived modular components.

4 Conclusion

The main goal of this thesis is stated in Chapter 1 as to reduce the required effort for integrating or adapting existing SLAM software to new applications. To achieve this goal, this work attempts to facilitate reuse and interchangeability of functionality between different SLAM systems by promoting the use of modular, composable components.

Within the field of SLAM there is little attention for the architectural organisation of SLAM systems in general and specifically SLAM as a software product. Previous efforts on the reusability of SLAM software indicate that the proposed solutions may run into difficulties when applied to real-world systems or different algorithms than covered by the corresponding research. The work presented in this thesis fills in this gap by considering both high-level SLAM theory and real-world SLAM implementations.

The following research questions were proposed in Chapter 1 for being covered by the research in this thesis:

1. What are the functions performed by current state-of-the-art SLAM systems?
2. What kind of information is required or processed in order to perform these functions?
3. In which way does the architecture of these SLAM systems influence the modularity and reusability of the associated functionality?
4. How can functional behaviour be structured in logical framework components and interfaces to facilitate reuse, interchangeability and adaptability?

The remainder of this conclusion consists of several Sections. In Section 4.1 the answers provided by this thesis to each of the research questions are discussed. This is followed by Section 4.2, which discusses the implications of the presented results within the field of reusable SLAM systems. Section 4.3 follows up on this, by explaining the implications for when the proposed framework is to be used in general. Several ideas for future work were raised during the development of this work, which are treated in Section 4.4. Section 4.5 discusses the limitations of this work and the presented framework.

4.1 Findings

The individual research question are discussed next, to go over the findings that contribute to answering these questions.

4.1.1 What are the functions performed by current state-of-the-art SLAM systems?

This question is targeted at gaining insights about the baseline of functionality that is required for any system which performs SLAM estimations. Section 2.1 and 2.2 approach this problem by going over the relevant SLAM theory and investigating the workings of two real SLAM implementations: RTAB-Map and ORB-SLAM2. Together, this paints a picture of the operations that are to be expected from a SLAM implementation.

The logical answer to the question about the performed functions follows directly from the theory, which states that (full-) SLAM estimates the trajectory $x_{1:t}$ and the map of the environment M. But by looking into the real-world SLAM implementations, insight into the "how" these estimations are made was gained.

Numerous methods for matching and comparing features and observations were identified. In both systems functionality for place recognition and loop closure detection also played a

prominent role. The functions that manage the different data sets for the algorithms and the attached storage pools were shown to be of great importance for the operation of both systems.

Deriving this baseline of functionality, as listed by Table 2.2 and 2.10, proved to be instrumental to answering the remaining questions.

4.1.2 What kind of information is required or processed in order to perform these functions?

The treatment of relevant theory in Section 2.1 did not extensively cover SLAM estimation algorithms. However, by the defining the control u_t and measurement z_t inputs, there is at least a vague idea about the information that is available to a SLAM system. Likewise, poses and constraints were expected to occur in some form throughout the analysed systems.

Again, the analysis in Section 2.2 provided much more insight in the dependencies between the operations in the different functional steps. These dependencies are associated with the information that is processed by the systems.

The most important observation made here is that along the expected information (features, poses, etc.), the SLAM implementations process a lot of information that is involved with managing the different data sets, identifying specific relations and orchestrating the execution of different functional steps.

The insights that follow from answering this question played an important role in the considerations for answering the next two research questions.

4.1.3 In which way does the architecture of these SLAM systems influence the modularity and reusability of the associated functionality?

Although the focus of this work is not targeted at modular software engineering practices, Section 1.4 provided some background information on aspects that influence the reusability of software in general.

By looking at these systems from a combined SLAM functionality and software engineering point of view, Section 2.3 answers the research question by evaluating the identified behaviour and dependencies within the analysed SLAM systems.

The most important observations that are made about the separation of functionality and responsibilities, are pointing out that these systems are suffering from leaky abstractions and may benefit from a critical look at the information that is stored and the place it is stored at. Of course, these observations come with the side note that the design of the analysed systems was most likely not targeted at the reusability and modularity of (parts of) the algorithm.

4.1.4 How can functional behaviour be structured in logical framework components and interfaces to facilitate reuse, interchangeability and adaptability?

The main contribution of this work follows from answering this specific question. The answer comes in two parts:

- The first part consists of Section 3.1, which developed the concepts and notation that facilitates a generalised way to discuss different SLAM systems.
- The second part to the answer is given by Section 3.2 and 3.3, which define and explain the use of a set of six modular components within the presented framework.

By considering a theoretical view on SLAM, along with two practical implementations, a direct comparison could be made between the functionality, requirements and concepts which follow from each of these contributions. As a result, the developed framework is a fitting compromise, based on the needs of both theoretical and practical implementations.

4.2 Theoretical implication

The framework presented in this work contributes to the field of SLAM by defining a strict set of concepts, entities and by distinguishing between the different correspondence-types. These definitions and the associated component types pave the way for capturing and describing the relevant details of different SLAM approaches in general. Furthermore, the framework considers SLAM as a combination of functional or algorithmic behaviour and a software product that is executed on a computational platform.

The modularity and reusability of SLAM algorithms specifically benefits from the presented definitions and components, by the boundaries that are defined between different components. At these boundaries, data dependencies due to the interfaces for these components are reduced by two mechanisms that follow from the proposed framework:

- The strict separation of the type of information that is available as output from a component.
- The definition of generalised data types and data structures, which can be used to implement interfaces.

The framework does not impose a strict separation between the conventional SLAM front-end and back-end operations. However, the separation between sensor-dependent operations of the front-end and sensor-independent operations of the back-end can still be facilitated by choosing an appropriate implementation for the components and structure for the composed algorithm. For example, by isolating the F-, C- and X-type components that are fed by the raw sensory inputs from the remainder of the system.

There are some noticeable differences and similarities with respect to the related work mentioned in Section 1.3.

- The "universal feature" used by Broenink (2016) is similar to the features used by this framework. His suggestion to allocate the responsibility for feature matching to the "feature module" has a similar effect as the pair of F- and C-type components proposed for feature extraction and matching in this work. Overall, the framework components in this thesis provide more granularity to implement different responsibilities.
- The definitions used for the input to and landmarks detected by the SLAM system deviate from those used by Minnema (2020). His "Idiothetic filter" component fulfils a similar role to the presented X-type component. However, the presented framework allows for a more flexible application of the different component types by using a flat hierarchy in which components are allowed to consume which ever type of information they need to fulfil their task. Sensor-independent operation is facilitated differently in the framework, by using alternative X-, F- and C- type component implementations for different sensory inputs.

4.3 Policy implication

Although there is no relation to policy making or legislation from this work, the proposed framework affects the relation between the two groups of stakeholders mentioned in Section 1.2. As mentioned in the introduction Chapter, this work specifically targets the users of SLAM systems who like to be able to reuse those systems for different applications.

The burden of developing reusable components for the construction of SLAM algorithms falls to the developers of those SLAM systems, instead of the users or system integrators. Where these categories of stakeholders overlap, there is no issue. However, SLAM developers that are not directly involved with having to support different applications, may need to be convinced

that making a modular design is worth the effort. Additionally, the threshold to apply the proposed framework must be made as low as possible.

The framework assumes that there is a need and dedication for reusability and modularity from the beginning of development. Especially if modularity is an afterthought, the required redesign may involve a huge effort. This approach causes fundamental issues for the development of new SLAM algorithms, as such a design process often does not fit well with experimental work where the requirements may not be set in stone. But even when trying out new ideas, it is beneficial that existing and stable components can be used to start from and keep the experimental code separated.

4.4 Future research

The development of the modular framework presented in this thesis raised some questions which are requiring further investigation to be answered. These are shortly discussed below:

- What would be the implications of having a SLAM algorithm with a dynamic structure, rather than a static structure at runtime? And can such a dynamic structure be leveraged to automatically adjust to changing operating conditions and applications or provide robustness against sensor failures? Such a dynamic approach can make use of the presented framework concepts to automatically assess alternative variants of components, or select different structures to produce the same output.

- Of a more practical nature is the question on how additional software tooling can aid the development of modular framework components and SLAM systems. Can these tools perform tedious and repetitive tasks such as providing interface specifications and component documentation? Can these tools provide a platform agnostic visualisation of the information that flows through the SLAM system? Reduce the development efforts by generating code and provide automatic composition and configuration of components?

- The development of new methods to benchmark, test and compare different SLAM implementations. Especially the dynamic behaviour of information that flows through the systems with respect to time is often unclear. It would be a tremendous improvement if the performance of SLAM systems can be specified in terms of robustness, timing and uncertainty like it is possible for control systems.

  Similarly, can the performance of a SLAM system be determined with respect to the velocity at which the robotic platform is moving, or can the algorithm be made responsive to the dynamics of the sensory inputs?

4.5 Limitations

During the development of the presented framework, a number of choices were made that may influence the applicability of the framework in its current state. These limitations are discussed next, in order to assess the issues they pose to the use of this framework.

- The scope of this work was limited to two visual, feature-based, pose graph SLAM methods. This may have caused a bias towards these types of algorithms within the framework. Further development to support other algorithms and sensor types is required to make sure that these systems also fit within the presented framework. However, the presented concepts are believed to be sufficiently generic to support feature-based methods with range finding type sensors and other algorithms like Kalman and particle filters.

- The framework presents a data-flow oriented view of SLAM systems, which in its current form does not describe the control flow of operations very well.

- The description of the analysed SLAM systems required some abstraction, as it is impossible to describe every tiny detail and operation. Although an extensive investigation of all interfaces, arguments and variables in the source code was performed, some behaviour or dependencies may have been overseen.

  Some parts of the analysed systems were explicitly ignored. However, it is strongly believed that most of this behaviour can be allocated to one or more components of the types specified by the framework without further alterations to the definitions.

- The localisation-only modes for RTAB-Map and ORB-SLAM2 were ignored and deemed to be "not-SLAM" due to lacking the mapping part. These localisation modes have a lot of functionality in common with the analysed SLAM algorithms, so there is potential for the framework to support these modes along the current SLAM mode.

  The framework may require further extension to support functionality like RTAB-Map's path planning mode.

- The strict separation of component types by the type of output they generate resulted in a moderately low-level description of the operations performed by SLAM systems. For algorithms that are more involving than RTAB-Map, this may result in a significant number of different components to present an accurate view of the workings of the system.

  If this proves to be a realistic issue in the future, the framework can potentially be extended with another level of abstraction. This level may support components which generate multiple output types at once, constructed by the currently presented set of components internally.

The limitations discussed above are not fundamentally breaking the components or definitions as given by the framework. Being able to discuss these issues in such a structured and detailed manner, shows that the framework is able to contribute to the language and concepts that are used to talk about SLAM systems. Due to the components facilitated by the framework, the consequences of any potential issue are easier discussed and reviewed.

4.6 Concluding statements

The work covered by this thesis allows to discuss the construction and workings of SLAM algorithms extensively. By structuring new SLAM systems according to the proposed framework, the role of each step in the system can be made very explicit. Hopefully, this contributes to a better understanding of and interesting new insights in the field of SLAM.

At the same time, the clarity and reusability provided by this framework lowers the threshold and required knowledge for users to apply SLAM algorithms for a wide range of applications.

Bibliography

Abdelhady, M. (2017), *Reuse-oriented SLAM framework using component-based approach*, Master's thesis, University of Twente.

Adrien Angeli, David Filliat, S. D. and J.-A. Meyer (2008), Fast and incremental method for loop-closure detection using bags of visual words, **vol. 24**, no.5, pp. 1027–1037.

Anguswamy, R. (2013), *Factors Affecting the Design and Use of Reusable Components*, Ph.D. thesis, Virginia Polytechnic Institute and State University.

Broenink, T. (2016), *On reusable SLAM*, Master's thesis, University of Twente.

Cesar Cadena, e. a. (2016), Past, present and future of simultaneous localization and mapping: toward the robust-perception age, **vol. 32**, no.6, pp. 1309–1332.

Cooling, J. (2003), *Software engineering for real-time systems*, Pearson Education Limited, Essex, England.

Ellery, D. (2017), *Writing reusable code for robotics*, Master's thesis, University of Twente.

Engel, J., T. Schöps and D. Cremers (2014), LSD-SLAM: Large-Scale Direct Monocular SLAM, in *European Conference on Computer Vision (ECCV)*.

Engel, J., J. Stueckler and D. Cremers (2015), Large-Scale Direct SLAM with Stereo Cameras, in *International Conference on Intelligent Robots and Systems (IROS)*.

Gálvez-López, D. and J. D. Tardós (2012), Bags of Binary Words for Fast Place Recognition in Image Sequences, **vol. 28**, no.5, pp. 1188–1197, ISSN 1552-3098, doi:10.1109/TRO.2012.2197158.

Geiger, A., P. Lenz and R. Urtasun (2012), Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite, in *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Grisetti, G. e. a. (2010), A tutorial on graph-based SLAM, **vol. 2**, no.4, pp. 31–43.

Gutierrez-Gomez, D. and J. J. Guerrero (2018), RGBiD-SLAM for Accurate Real-time Localisation and 3D Mapping.

Horn, B. K. P. (1987), Closed-form solution of absolute orientation using unit quaternions, **vol. 4**, no.4, pp. 629–642.

Kaess, M., H. Johannsson, R. Roberts, V. Ila, J. Leonard and F. Dellaert (2011), iSAM2: Incremental smoothing and mapping with fluid relinearization and incremental variable reordering, in *2011 IEEE International Conference on Robotics and Automation*, pp. 3281–3288, doi:10.1109/ICRA.2011.5979641.

Labbé, M. and F. Michaud (2011), Memory management for real-time appearance-based loop closure detection, in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Minnema, J. (2020), *Towards component-based, sensor-independent and back-end independent SLAM*, Master's thesis, University of Twente.

Mur-Artal, R. and J. D. Tardós (2017), ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras, **vol. 33**, no.5, pp. 1255–1262, doi:10.1109/TRO.2017.2705103.

Pire, T., T. Fischer, J. Civera, P. De Cristóforis and J. Jacobo berlles (2015), Stereo Parallel Tracking and Mapping for robot localization, in *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*, pp. 1373–1378, doi:10.1109/IROS.2015.7353546.

Qin, T., P. Li and S. Shen (2018), VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator, **vol. 34**, no.4, pp. 1004–1020.

Rainer Kümmerle, e. a. (2011), g^2o: A General Framework for Graph Optimization, in *IEEE International Conference on Robotics and Automation*.

Raúl Mur-Artal, e. a. (2015), ORB-SLAM: a Versatile and Accurate Monocular SLAM System, **vol. 31**, no.5, pp. 1147–1163, doi:10.1109/TRO.2015.2463671.

RobMoSys (2017).
https://robmosys.eu/

Schlegel, D., M. Colosi and G. Grisetti (2018), ProSLAM: Graph SLAM from a Programmer's Perspective, *2018 IEEE International Conference on Robotics and Automation (ICRA)*, doi:10.1109/icra.2018.8461180.
http://dx.doi.org/10.1109/ICRA.2018.8461180

Schneider, T., M. T. Dymczyk, M. Fehr, K. Egger, S. Lynen, I. Gilitschenski and R. Siegwart (2018), maplab: An Open Framework for Research in Visual-inertial Mapping and Localization, *IEEE Robotics and Automation Letters*, doi:10.1109/LRA.2018.2800113.

Sebastian Thrun, e. a. (2005), *Probabilistic robotics*, The MIT Press, Cambridge, Massachusetts.

Sivic, J. and A. Zisserman (2003), Video Google: a text retrieval approach to object matching in videos, in *Proceedings of the ninth IEEE International Conference on computer vision*.