A Component-Oriented Approach to Simultaneous Localization and Mapping

Mickey Ristroph

May 2, 2008

Abstract

The simultaneous localization and mapping (SLAM) problem is central to many mobile robots. The construction of a map of the local environment and the localization of the robot in that map must be accomplished in an incremental manner, even in the presence of significant error and uncertainty in sensor data. Further, the cyclic data dependency presents a challenge that does not lend itself to robust solutions. Solutions to the SLAM problem have been a major success in robotics research in the past 20 years. Existing solutions fully embrace the self-referential nature of the problem.

There is a certain art to designing and tuning systems that achieve stability and convergence properties. Most robotics platforms implement a customized version of SLAM with modifications to improve performance, given certain assumptions and a priori knowledge of the environment.

Borrowing techniques and tools from the parallel composition research community, we aimed to design a robust, extensible, and efficient framework for SLAM solutions using a component-based architecture. This begins with a domain analysis, characterizing the breadth of existing solutions and factoring the logical function of various components in a modular way. We then define the interfaces for these components, provide implementations, and connect them in a data flow or dependency graph.

The final implementation presented supports, in theory, particle filter localization and can benefit from automated parallelization. However, it has not yet run successfully at the time of writing. The reasons for this include missing key features and non-working implementations of purported features in the tools used, PCOM^2 and CODE. These limitations are described in detail and motivated, and should serve as justification for future work in parallel component composition research. Unfortunately, these are the only presentable results of the research done here at this time.

1 Introduction

1.1 Overview

Ever since we started using computers to solve even moderately complex problems, managing the complexity of the software used to solve those problems has been immensely important. Computer scientists have invented a multitude of approaches, each which allow us to reason about, create, test, and perhaps even verify the correctness of that software. An approach is some collection of conceptual models, design patterns, styles, conventions, languages, and logic

systems that can be used to formalize the proposed solution into something a computer can understand.

First, we will describe in detail one of those complex problems we use computers to solve. It's called Simultaneous Localization and Mapping, or SLAM, and although it is similar to problems in other fields, we will approach it from the domain of mobile robotics. The first papers discussing the problem from the robotics perspective were published in 1986[15].

Then, we will describe one of those approaches to solving complex problems. The approach can not be as well delineated as the problem, but is usually called a component-oriented or component-based approach. The first papers discussing the approach were published in 1968[12].

The remainder of the paper will then describe the specific tool used, PCOM^2, and present the results of an attempt at applying the component-oriented approach to the SLAM problem using that language. We conclude with a discussion of the lessons learned from that attempt.

1.2 Localization and Mapping

1.2.1 Mobile Robots

A *robot* is a computer-controlled device that interacts with the real, physical world. To do this, it has some sensors that perceive things about it's environment, and actuators to change things about it's environment. Under this general description, a modern computer-controlled thermostat on an air conditioning unit counts as a robot, as does your automobile's cruise control, and more obvious examples, like the Mars Rover or the robots used on industrial assembly lines.

A *mobile* robot is one that moves around it's environment. Reasons for doing so vary, and may include transporting things, making changes to the environment over a large area, or gathering information about a large area.

Whatever the reason, since the robot is moving, it is usually important that the agent controlling the robot know where the robot is. It is also usually important to know where other things are, such as obstacles. The following subsection discusses ways to acquire those two pieces of information and how they are related.

1.2.2 Localization

The process of determining the location of a mobile robot relative to some external frame of reference is called *localization*. Our analysis considers only the localization problem of a single robot in a static environment. Further, we will be concerned with only the passive portion of localization–that is, there is no discussion of any control algorithms related to minimizing localization error.

Location is a time-dependent state variable. The dimensionality will depend on the application. Is it also important to note that calling this piece of state "location" can be misleading. In practice, it usually consists of not only a two or three-dimensional location in some Cartesian coordinate frame, but also rotations along axes defined by the vehicle's own reference frame, namely, pitch, yaw, and roll. Section 1.2.5 will explore this in more detail, including a partial formalization. The remainder of this section discusses localization in more loose, intuitive terms.

The localization problem is then the problem of estimating that location and orientation. There are three important sources of information used in making this estimation.

First, sensor data will certainly be used. This could include measures of pose directly, laser range finders, sonar readings, tactile sensor readings, images from cameras, signal strengths

from antennae, or any number of other things. Sensors are usually not perfect, and for any sensor used, we ought to have some understanding of the nature of the noise.

Second, mobile robots usually have some control parameters that are set by the controlling agent. For example, we might be set a desired forward velocity and a desired turning rate. Just like sensors, the actuators that execute that command are probably not perfect, and we need a similar understanding of how good the control of the robot's motion is.

Lastly, the process will use information already known about the environment. We may know the locations of fixed objects that are perceivable by our sensors. We will refer to the collection of information known about the environment as a *map*. Although we will explore the nature of this information in more detail, most pure localization techniques assume certainty in the map information.

Given the uncertainty in our sensors and control, most commonly used localization algorithms are probabilistic. At the most generic level, all localization algorithms determine a likely or the most likely location and orientation given the history of sensor data, control parameters, and the map.

1.2.3 Mapping

In addition to localization, if the mobile robot will potentially interact with environment, the controlling agent will probably need to know the location of other things in the environment. As mentioned in the previous section, a map is such a collection of information known about the environment.

There are many different things that could be known, and many different ways of storing that information. There are metrical maps, which store locations of things, and topological maps, which store information about the connectedness of spaces. Maps may take the form a discreet grid, or some vector format. The richness of the information in the map may vary. At the simplest level, a map may only contain binary information about whether a portion of the environment is occupied. However, it could also contain things like temperature, elevation, or even a functional aspect of the area, like where power outlets are.

In efforts to keep things manageable, consider only two dimensional maps, defined over a fixed region of space, using a grid, and storing only information about the whether the region of space corresponding that that grid cell is occupied. We will cleverly call this kind of a map an *occupancy grid*. Despite the simplifying assumptions, this kind of map is very effective and widely used in mobile robots.

Sensors like tactile switches, laser range finders, sonar, or cameras provide information about the environment in the robot's frame of reference. In order for a mobile robot to make a map without resetting the map after each reading, there must be some way to put different readings in a common frame of reference. The information to perform this transformation is change in the location and orientation of the robot, the same information the localization process determines.

However, if the location and orientation were exactly known at all times, all sensor readings can be transformed to a common coordinate system, out of which a map can be built. The challenge lies in the cyclic information dependency of the localization and mapping problems.

1.2.4 The Challenge of SLAM

Accomplishing both these tasks simultaneously seems paradoxical. We must use our location and sensor data to build a map, and yet we are using the map and sensor data to determine our location. What are we to do about the "chicken and egg" nature of the problem?

Most solutions fully embrace this cyclic dependency, and solve it by making some initial assumption. The map and location information are bootstrapped from that, making incremental

improvement. For example, if we do not know our initial location, we can assume we are at the origin of some coordinate system, and figure out where we are relative to our starting position in subsequent iterations. There are even provable convergence properties regarding our process for doing this, under certain assumptions[6].

1.2.5 Partial Formalization

SLAM is one of the most well-studied problems in robotics. The mathematical underpinnings of the problem have been well formulated using discrete time Markov processes and Bayesian probability theory. This section is a review the formalization of SLAM that serves as the justification for most algorithms and implementations.

The *configuration* of a robot refers to the coordinates in three-dimensional space, and the rotational angles about each of the three axes. Localization is the process of determining some subset of these variables. *Pose* refers to the x and y coordinates, as the rotation about the z axis. This is not to be confused with *position*, which usually refers to the location information without any rotational components[17].

Pose is usually sufficient information for control of mobile robots whose motion is restricted to a surface, as it usually sufficient to perform the necessary translational and rotational transformations to bring all sensor data into a common frame.

Under this simplification, the localization problem can be stated as follows. Ideally, we would like to know the exact pose of the robot at every time t, given by[15]:

$$x_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

Given the uncertainty, we instead attempt to estimate the probability of each possible x_t, given a map m, the history of sensor data $z_{1:t}$, and the history of control parameters $u_{1:t}$:

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

Depending on the algorithm, this "estimation" may consist of only a single x_t for which the probability is high. Also, note that as stated, the estimation of x_t does not depend on information gathered after t. This makes it possible to calculate at time t, and can therefore be done live, or *online*[17]. Some localization algorithms use information gathered after time t in the estimation, which can useful for figuring out where the robot was, but cannot be used live. These variants are called "offline".

If the estimation of $p(x_{t+1})$ depends on $p(x_t)$ but is independent of estimations at times $t-1$ and before, it satisfies the *Markov property*[2], and this called Markov localization.

The mapping problem can be described in similar terms. Ideally, we would like to know the exact map of the environment. Using probablistic techniques, we instead estimate the probablity of a map at time given all the pose, sensor, and control information gathered previously:

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

Combining these two ideas, the SLAM problem, in the most general form, is then estimation of a joint probability distribution of both the pose and the map, given the history of sensor and control data:

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

1.3 Component-Oriented Architectures

1.3.1 Overview

The idea behind the component-oriented approach to software design was first proposed by Douglas McIlroy in 1968[12]. The proposal came at a time when the idea of having good software architecture was largely unheard of. It calls for many improvements in software engineering that are now fairly well-understood and widely in use, such as good encapsulation and and improved reusability of code.

It is important to note that the design strategies of the component-oriented approach described here and in the cited literatures is not technically well-defined. The particular language we will use does precisely formalize the approach, but we will start with a high-level description of the idea behind component-oriented design.

Much of the inspiration comes from the a desire to make software engineering more similar other engineering disciples. McIlroy observed "the software industry is not industrialized"[12], and sought the software analogy of interchanged parts introduced by industrialization in other engineering fields.

For that reason, the component-oriented approach is easily understood by analogy to electrical engineering. One ought to be able to build a software system in a similar manner to how one builds an electrical system–out of pre-built components. There are very basic components, like transistors, resistors and capacitors, and there are composite components, also called integrated circuits (ICs), that factor some of the common tasks. ICs vary in complexity, from a simple timer circuit, to large microprocessors or memory banks. The interface to these components creates a convenient level of abstraction, and must be exactly specified. It may take the form of a pin-out diagram, or an instruction set architecture, in the case of processors. When viewed through this analogy, the schematic diagram of a circuit specification is similar to the data-flow graph of a software system.

So what are some examples of components in modern component-oriented systems? Database servers are a commonly used example[1, 9]. They provide a well defined service through a well-defined interface, usually a variant of SQL. Other examples include messaging and queueing libraries, and many web services, such as the service provided by the Google Maps API.

One illustrative example of a component-oriented system with which many are familiar is the UNIX pipeline. The original implementation, in face, was written by McIlroy. There are a collection of simple programs, such as `cat`, `grep`, `sed`, and `awk` that each do a relatively simple task, but can be easily composed on the command line with the `|` symbol to created more complicated programs. Note that this system supports only serial composition, hence, it is a "pipeline", whereas many systems support more arbitrary graphs.

1.3.2 Defining a Component

Definitions of what a "component" is vary widely. Consensus exists only at a high level: components are reusable, composable, encapsulated, and are units of deployment and versioning[5, 9, 14, 10, 1]. They have well-defined interfaces through which they provide some service.

There are two important pieces to a component framework: the specification and composition. The component specification portion allows the user to define components, where as the composition portion enables the assembly of many components into a program or system.

1.3.3 Component Specification

The most important part of component specification is the specification of the *accepts* interfaces, through which the component is invoked, and the *requires* interface, through which the component invokes other components[11]. This specification can occur with varying levels of

formalism. In a 2002 analysis of a literature survey, Norby and Blom identify five levels of semantic formalism[5]–we will discuss three of those.

At lowest level, there can be no semantics, in which the specification does no include any description of the operation of the component. At this level, components are specified in a purely syntactic way. This can be understood as function signatures languages like C. The type of the parameters and returned value are specified, but there no description of meaning other than the convention of descriptive variable names.

Somewhere in the middle are intuitive semantics, in which the syntactic specification is supplemented by a text description of the interface and component functionality. This could consist of a comment warning the user that behavior is undefined for parameter values in a certain range.

At the highest level, components are formally specified, meaning they can be proved to have consistent and sound semantics. This also allows an automated system to reason about the functionality of the component, and prohibit illegal compositions.

It is at this highest level of semantic specification where two big benefits of a component-oriented system, automated composition and verifiable composition, become realizable. For this reason, we will aim for this level for semantic formalism.

1.3.4 Component Composition

Component composition is how components are assembled to make other components, programs, or systems. This can often take the form of writing "glue code", as is the case with the previously mentioned UNIX pipeline example[5]. Ideally, glue code should be as simple and easy to write as possible, leaving most of the work for the components.

Many systems also provide a graphical way to compose objects. In the interface, this often takes the form and dragging and dropping components in place, and drawing connections between then. Graphical composition can occur at various levels of semantic formalism, but is aided by a higher degree of formality. The graphical system prohibit unexecutable compositions with sufficient formal semantics.

Lastly, given a specification of a desired task, automated composition is possible with the highest degree of formalism. A system that can reason about the functionality of components can possibly decide how to arrange components to achieve a desired result.

The ease of component composition is one metric of the quality of a component-oriented system. This is the part that will enable the quick construction of a large family and variety of systems.

1.3.5 Advantages

As mentioned, a well-designed composition systems allows for the rapid creation of a class of similar solutions. However, there is another very important potential benefit. Because the data-flow or dependency graph of the components is explicitly given, automated systems can identify components that can execute in parallel, and do so on separate cores, processors, or even machines. Automating parallelization of code is especially important as the multi-core and multi-processor machines are becoming the status quo.

1.3.6 Disadvantages

Formalizing the interfaces of the various pieces of a system, while a useful exercise for any architect, is a difficult task. One must weigh the costs and benefits of such an exercise. Since the advantages come with formally specified interfaces, the architect must decide to accept

additional effort in specification in order to reap the benefits of ease of composition. For systems not needing extensive reconfiguration, this may not be worth it.

1.3.7 Examples of Component-Oriented Frameworks

There are many component-oriented frameworks, the most well-known of which include CORBA[19] and Java Beans[3]. However, most relevant to this work are two that are specific to the mobile robotics domain, Player[4] and CARMEN, and another that provides the highly formal component semantics necessary for the advantages we are seeking, PCOM^2[11].

2 Method and Results

2.1 Domain Analysis

The first step of taking a component-oriented approach is a domain analysis. This consists of identifying the logical functions present in each solution, identifying which of those are shared between different algorithms, and classifying the various implementations.

Most SLAM implementations consist of fairly well separated localization and mapping components, as well as one or several components that handle reading the sensor data.

There are several aspects of the localization that may vary between systems. Some include the availability of an initial location, availability of control parameters, active control for minimizing localization error, and online or offline location estimates.

Similarly, in the mapping component, we need to consider the availability of an existing map, if and how dynamic aspects of the environment are modeled, the type and manner of information stored, such as grid-based or feature-based map, or some other representation, and what information is being stored, such as occupancy information, or altitude information.

2.1.1 Survey of Well-Known SLAM Algorithms

The Graph-SLAM algorithm, as presented by Thrun and Montemerlo[18], consolidates the work of many researchers in offline SLAM on feature-based maps. Given our emphasis on creating a system for live use on robotics platforms, we will not explore any offline approaches in detail.

The Extended Kalman Filter approach to online SLAM using a feature-based map was published by Cheeseman and Smith[15] and was the first to approach the problem in robotics. However, the Extended Kalman Filter has two limitations. The mathematical justification relies on Gaussian noise, and the maintenance of the covariance matrix equates to roughly quadratic complexity.

The computational complexity issues of the EKF approach for online SLAM with feature-based maps are addressed with the Sparse Extended Information Filter approach[17], which attempts to reduce the complexity to linear by "sparsifying" the matrix. This involves only paying attention to nearby features, and ignoring the relationship between features that are believed to be distant.

The information theoretic approaches used in feature-based SLAM algorithms does not transition well to grid-based maps. Instead, particle filters dominate the localization techniques used in grid-based SLAM.

Particle filters maintain a set of hypotheses of possible poses, or possible poses and maps. Each particle can be assigned weights using the grid-based map according to a measurement model. FastSLAM is a popular implementation of this[13], although the algorithm also has

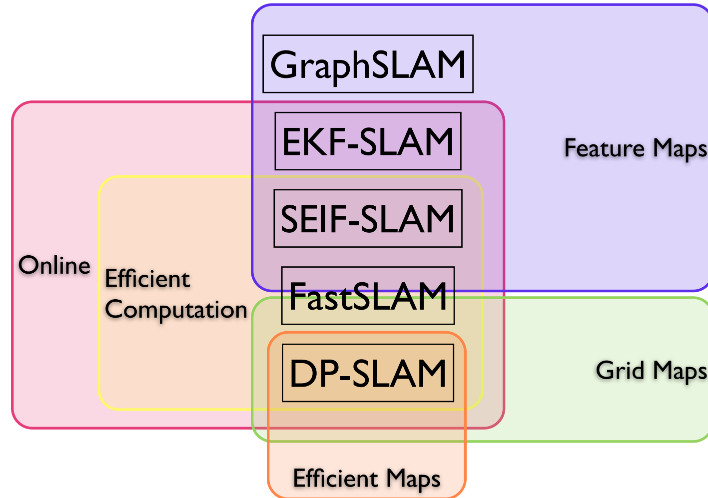

Figure 1: Classification of well-known SLAM algorithms

feature-based variants. The set of particles is the estimation of the joint probability distribution of the pose and the map.

A disadvantage of this is the large number of maps needed to be stored. Another SLAM algorithm, DP-SLAM[7], arranges the map data structures so that different hypotheses still share in memory the portions of the map which are not changed.

These results are summarized in Figure 1.

2.2 Choosing Components and Interfaces

At the highest level, we will decompose SLAM into five components: Initialization, the Sensor Handler, the Mapping Service, the Localization Service, and a Map Visualizer. The data flow diagram common to any SLAM algorithm is given in Figure 3. The control flow diagram is given in Figure 2.

As discussed previously, there are two alternative semantics for component invocation: data flow and call-return. Currently, PCOM[2] only supports data flow semantics[11], so that will be focus of this analysis. Invocations with data flow semantics obviously require an analysis of the data or control flow of the algorithm, whereas call-return semantics would require an analysis of the dependency graph.

Because of this, the localization service must operate on the map without repeatedly passing control to the mapping service, as map queries would come from such a variety of contexts that returning control to the calling context would be impossible. This limitation dictates portions of our interface. The initialization component must send the initial grid and particles to the localization service, as the localization service must have a copy of the grid internally. Originally, we designed the localization service to invoke the mapping service for each query or update, in a manner similar to large systems components, like database servers, or object invocation. This would enable the localization to occur in a manner ignorant of how the map is implemented. However, that would use the call-return semantics, in which the result is sent

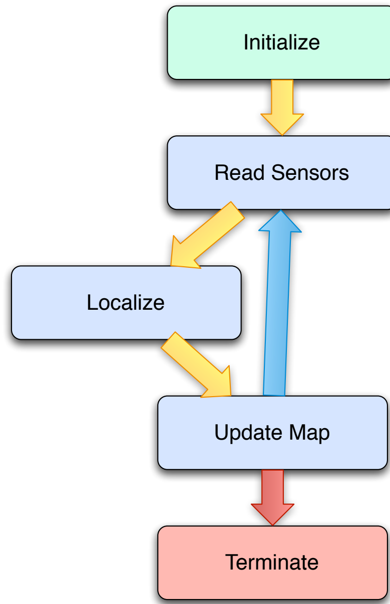

Figure 2: Serial control flow for SLAM algorithms

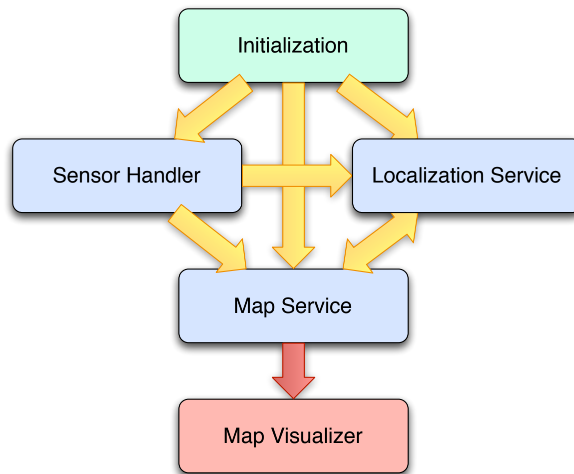

Figure 3: Basic data flow of SLAM algorithms

to the invoking component. As this is impossible in PCOM^2. Instead, the invoked component must specify where the result is sent, according to data-flow semantics.

Similarly, the initialization must send the initial grid and path to the mapping service, and a signal to begin to the sensor handler. The sensor handler will send the new sensor data to the localization service, which will send the best estimate of pose to the mapping service, along with a copy of the sensor data. The mapping service will update the map, append the pose to the path it is maintaining for analysis purposes, and send the localization service the updated map. It will signal the sensor handler that it is reading for additional data.

The map service can also, at any time, signal the map visualizer to draw the map. In our implementation, this happens at the end of sensor data, as signaled by the sensor handler. These data and control flow specifications are shown in Figure 4.

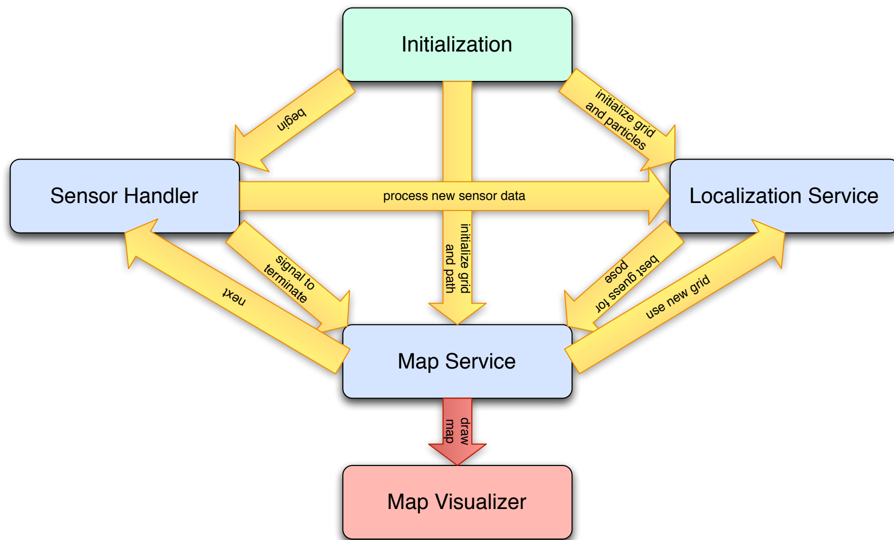

Figure 4: Control and data flow of SLAM for Occupancy Grids

For particle filter approaches, we may reap the benefits of parallel composition by noting the calculation of the probability of hypothesis is independent of the others. We can create components that break the up the localization in three components: one that distributes the particles among several parallel components, each of evaluates the measurement model for that hypothesis, and a third component that collects the evaluated particles. This is shown in Figure 5. These components are invoked through a wrapping component that provides the same interface as the original serial version. In theory, if we are using a parallel composition component framework such as PCOM^2, this arrangement can be compiled for serial computation, in which case the functionality is the same, or a parallel environment, in which the computation can we spread between several cores.

2.3 Implementation

The implementation of the SLAM with particle filter localization is complete, however, it has not been successful to date. There numerous issues with the language that we have either been

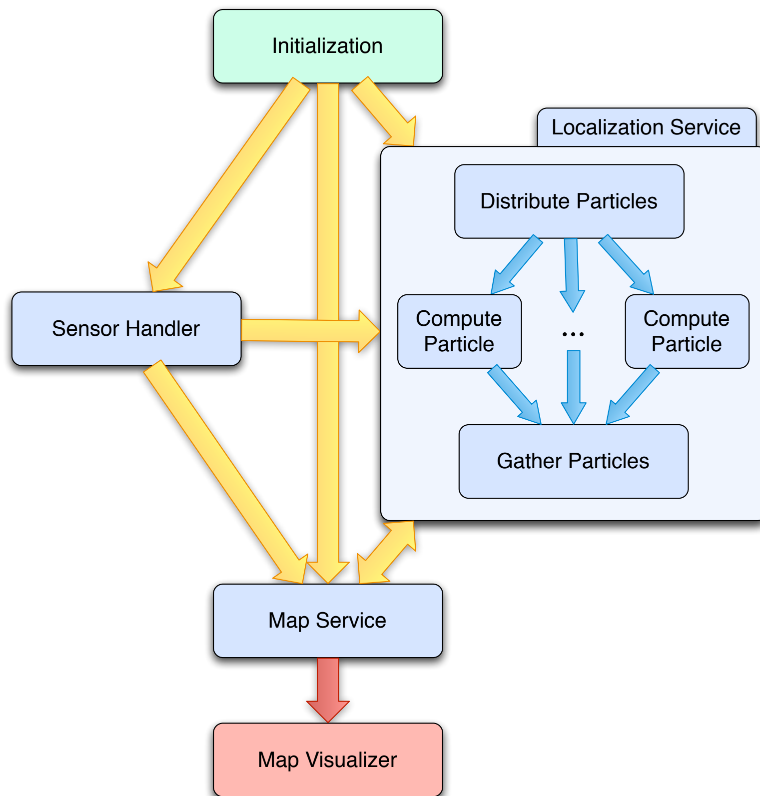

Figure 5: Control and data flow of Particle Filter SLAM for Occupancy Grids with Parallelization

unable to work around, or have created awkward architecture designs and slowed development.

The first limitation is the missing call-return semantics, which makes the componentization the results from the domain analysis impossible. Ultimately, the logical functions of the components of slam are services that return values, such as the map. Given this limitation, we are in unfortunate situation of having our language dictate our architecture–not simply expressing our ideal architecture in the language.

A second limitation is the `&&` operator, which is for requiring multiple transactions of a component be invoked before executing, simply does not seem to work as described in the paper[11]. We would like to use to execute the localization routine once new sensor data *and* a new grid is received, but it does not function.

A latest limitation is the apparent inability to pass nested data structures between components while retaining a copy. The `copy_retain`, which does work for more primitive structures, doesn't work for nested data structures. If there is a way to do, it is unknown to anyone currently working with language, or the original author.

While not of interest to the research community, there are seriously blocking issues involving the maturity and maintenance of the code: a lack of a build system, sparse and out-of-date documentation and examples, a lack of helpful compiler error message, difficulty in debugging due to unsupported string literals, making adding simple print statements an arduous task, and many similar other problems. These are important because they do represent a real barrier to one goal of the original project–to develop a component-based system for SLAM with tools that have semantic formalism. These problems do need to addressed and for that reason are mentioned here, however, do not need to be discussed in more detail.

3 Discussion

3.1 Limitations in Scope

Many aspects of both SLAM research and component-oriented systems were not explored in this analysis. For example, we did not explore mapping systems that model a dynamic environment, particle filters that support a varying number of particles through KLD sampling[8], or map-building in a multi-agent system[16]. All these things are reasonable extensions, and should be incorporated if the parallel composition tools enable component specification and composition consistent with these algorithms.

There are also many other component-oriented systems that we do not attempt to implement SLAM in, such as CORBA[19] and Java Beans[3].

3.2 Discussion and Future Work

The field of mobile robotics has enjoyed some of the benefits of a component-oriented approach taken towards software architecture[4]. However, many of the true advantages of the approach will be realized when the tools use an increased level of semantic formalism in component specification. These benefits include an increased family of problems solved using the library of components, performance benefits through, automated parallelization, and most importantly, easier composition of a larger set of systems.

To this end, the next step from the robotics community is to understand and introduce semantic formalism into the existing component frameworks, such as Player and Carmen, borrowing from tools like PCOM[2].

Similarly, the fields of component systems and parallel composition have enjoyed some of the benefits of their tools being used in other domains, but we have identified some limitations that will slow or inhibit pervasive use of those tools. These limitations include an incomplete

or unclear separation between component specification and composition, and unimplemented call-return component invocation semantics. In addition, many of the parallel composition tools used and developed in the research community lack the maturity of programming and system architecture tools, as discussed in Section 2.3.

Therefore, the next step from this community is to understand why these limitations are important to robotics system architects and other potential users, refine these tools, and in particular, extend the component invocation semantics to enable a style roboticists will find more natural.

If researchers in both of these communities work in these directions, there is hope for meeting in the middle and having a robust, extensible, verifiable, and highly functional component-oriented mobile robotics framework.

References

[1] U. Abmann. *Invasive Software Composition.* Springer, 2003.

[2] A.T. Bharucha-Reid. *Elements of the Theory of Markov Processes and Their Applications.* McGraw-Hill, 1960.

[3] B. Burke and R. Monson-Haefel. *Enterprise JavaBeans 3.0 (5th Edition).* O'Reilly Media, May 2006.

[4] T. H. Collett, B. A. MacDonald, and B. P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proceedings of the Australasian Conference on Robotics and Automation*, 2005.

[5] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems.* Artech House, 2002.

[6] M. W. M. G. Dissanayake, P. M. Newman, H. F. Durrant-Whyte, S. Clark, and M. Csorba. A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on Robotic and Automation*, 17(3):229–241, 2001.

[7] A. Eliazar and R. Parr. Dp-slam 2.0. *IEEE International Conference on Robotics and Automation*, 2004.

[8] D. Fox. Adapting the sample size in particle filters through kld-sampling. *International Journal of Robotics Research*, 22, 2003.

[9] G. Leavens and M. Sitaraman. *Foundations of Component-Based Systems.* Cambridge University Press, 2002.

[10] Z. Liu and H. Jifeng. *Mathematical Frameworks for Component Software.* World Scientific, 2006.

[11] N. Mahmood, G. Deng, and J. C. Browne. Compositional development of parallel programs. *Lecture Notes in Computer Science*, pages 109–126, 2003.

[12] D. McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[13] Michael Montemerlo. *FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association.* PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, July 2003.

[14] M. Norris, R. Davis, and A. Pengelly. *Component-Based Network System Engineering.* Artech House, 2000.

[15] R. C. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainly. *International Journal of Robotics Research*, 5(4):56–68, 1987.

[16] S. Thrun. An online mapping algorithm for teams of mobile robots. *International Journal of Robotics Research*, 20(5):335–363, May 2001.

[17] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[18] S. Thrun and M. Montemerlo. The GraphSLAM algorithm with applications to large-scale mapping of urban structures. *International Journal on Robotics Research*, 25(5/6):403–430, 2005.

[19] N. Wang, D. C. Schmidt, and C. O'Ryan. Overview of the corba component model. pages 557–571, 2001.