Toward Adaptation and Reuse of Advanced Robotic Algorithms

Christopher R. Baker

CMU-RI-TR-11-09

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics.

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

04/2011

Thesis Committee:
John Dolan, Chair
David Wettergreen
Jonathan Aldrich, Institute for Software Research
Issa A.D. Nesnas, Jet Propulsion Laboratory, California Institute of Technology

This research was supported through the General Motors/Carnegie Mellon University Autonomous Driving Collaborative Research Laboratory.

Abstract

As robotic systems become larger and more complex, it is increasingly important to compose them from reusable software components that can be easily deployed in novel systems. To date, efforts in this area have focused on device abstractions and messaging frameworks that promote the rapid and interoperable development of various perception, mapping and planning algorithms. These frameworks typically promote reusability through the definition of message interfaces that are sufficiently generic to cover all supported robot configurations. However, migrating beyond these supported configurations can be highly problematic, as generic data interfaces cannot fully capture the variability of robotic systems.

Specifically, there will always be pecularities of individual robots that must be explicitly coupled to the algorithms that govern their actions, and no single message or device abstraction can express all possible information that a robot might provide. The critical insight underlying this work is that while the information that contributes to a given algorithm may change from one robot to the next, the overall structure of the algorithm will remain largely undisturbed. The difference is made in comparatively small details, such as varying individual weights or thresholds that influence the results of, but do not otherwise interfere with, the algorithm's "main" calculations.

This work proposes that exposing a few such points of variation in a given robotic algorithm will allow the modular treatment of a wide array of platform-specific capabilities. A corresponding design methodology is proposed for separating these platform-specific "supplemental effects" from a reusable, platform-independent "core algorithm". This methodology is evaluated through case studies of two distinct software systems, the first drawn from the realm of autonomous urban driving, and the second from the domain of planetary exploration. The central contributions of this work are:

- A nomenclature and corresponding guidelines for discriminating between platform-independent "primary" data and platform-specific "supplemental" data;
- Quantified costs and benefits for two technical solutions to isolating the corresponding core algorithms from their supplemental effects;
- A classification of typical segments of advanced robotic algorithms that can be affected by platform-specific data;
- A set of principles for structuring such algorithms to simplify the accommodation of future supplemental effects.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Robotic software systems are notoriously complicated, costly to develop, and difficult to reuse, in whole or in part, from one robot to the next. Many of these issues arise from the nature of complex software systems, especially the difficulty of subdividing a large system into a collection of smaller components, and managing issues such as synchronization and messaging between those components. Research and practical efforts in these areas have yielded a number of excellent robotic application frameworks and associated toolkits, such as CARMEN[37], Player/Stage[11], CLARAty[38] and the emerging ROS[46]. These generally focus on reuse in terms of a component's expected input and output data such that, as long as all of its specified inputs are available, that component can be reused in any number of applications generated using the same framework.

Unfortunately, these conditions, of using the same framework and providing *exactly* the same data, are often more constraining than openly admitted by proponents of component-based robotics. The consequences of violating these conditions are that some components must be invasively modified, i.e., their source code must be acquired, understood, and altered to accommodate the details of a new robotic platform. These consequences are widely recognized as undesirable and "to be avoided when possible", but there is otherwise little direct discussion of how, precisely, to avoid such modifications, or what to do when they simply cannot be avoided. This leaves developers with very little guidance as how to structure or modify existing components for use on other robots, resulting in ad-hoc adaptations that are difficult, error-prone, and can leave the adapted component in a much worse state for the "next" modification[1].

Analysis of the "lessons learned", or "difficulties encountered", in the development and application of component-based architectures can provide critical insights into the specific challenges of incorporating existing robotic software components into novel systems. For instance, the MARIE[12] framework successfully demonstrated the integration of components from several of the aforementioned toolkits, but their approach relies on the assumption that the data provided by one component are semantically identical to the data expected by any consuming components. This is not always, or even often, the case, as alluded to by one particular "difficulty encountered" by the authors of MARIE: that components often contain "hidden assumptions" about the robots they were originally written for. The result-

[1] This phenomenon is often described as an accumulating "calcification" or "brittleness"[13] of those components relative to further alterations of the surrounding system.

ing conflicts could only be rectified through significant modification of those components' inner workings, which were "tedious and error-prone", and required detailed understanding of the differences between the original and target platforms.

This difficulty is consistent with discussion surrounding CLARAty[39], which notes that even within the fairly narrow realm of planetary (Mars) rovers, variations in sensor selection and placement, mobility and actuation capabilities, power and communication architectures, and even mission context can have ripple effects through many of the software components that operate any given robot. These ripple effects make it difficult to isolate common functionality in reusable components, especially when attempting to define the interfaces between them, for which "neither the union of all possible capabilities, nor their intersection" were satisfactory.

Whereas these and other previous efforts focus on isolating platform variability behind increasingly generic data representations, this work instead embraces the idea that for some robotic software components, no single input specification can encompass the entire range of data that are available and relevant to the embodied algorithms. That is, there is always the possibility that migrating to the "next" robot will add, remove or alter the meaning of individual data provided to such algorithms, requiring introduction, excision, or modification of the corresponding effects of those data.

The critical insight underlying this work is that while the information that contributes to a given algorithm may change from one robot to the next, the overall structure of the algorithm will remain largely undisturbed. The difference is made in comparatively small details of the algorithm, such as varying individual weights or thresholds, that influence the results of, but do not otherwise interfere with, the algorithm's "main" calculations.

Thesis Statement

This work proposes that:

> **For many robotic algorithms, there is an important distinction to be made between the *primary* data that drive the *core* implementation of the algorithm and the *supplemental* data that enhance the core algorithm to exploit platform-specific details. Through the use of modern software design techniques, it is possible to isolate such platform-specific *supplemental effects* from the platform-independent *core algorithm*, yielding an artifact that is both more reusable across and more adaptable to a wide variety of robotic systems.**

For the purposes of this thesis, a "point of variation" can be thought of as a small segment of source code for a given robotic algorithm that has to be modified to accommodate platform-specific capabilities. These can be as simple as isolated if-then-else logic expressions, and are often deeply embedded, and thus difficult to locate and understand, in typical "reusable" robotic software components. "Exposing" these points of variation refers to the rearrangement of the structure of an algorithm to highlight the ways that it may be influenced by platform-specific capabilities, such that future developers can easily locate and understand how the algorithm may be adapted to suit their specific needs. The "modular treatment" of these adaptations refers to keeping individual robot-specific variations of an algorithm in separate classes, objects, etc., from the generic, reusable implementation of that algorithm. This allows dependencies on individual platform-specific capabilities to be introduced or removed by adding or excluding the corresponding classes, instead of having to make surgical alterations to the otherwise "reusable" parts of an algorithm.

The ultimate goal of this work is to equip future developers with tools and insights that will help them design and implement advanced robotic software components that can be easily adapted to the details of novel systems in this manner. Taking this "developer's perspective", the critical questions answered by this thesis are:

1. Without knowing about all future robots in advance, how can I discriminate between "platform-specific" and "platform-independent" capabilities?

2. What constitutes a "likely point of variation", and how do I identify them in my own algorithms?

3. Is the "modular treatment" of individual platform-specific variations technically feasible?

4. What are the tradeoffs that I should be aware of when considering this approach in my own systems?

The answers to these questions mark the primary contributions of this thesis, which explores the issue of platform-specific variations of otherwise generic algorithms through two case studies of existing software components. This begins in Chapter 2 with a breakdown of the most basic types of algorithmic adaptation that are compelled by changes in platform-specific capabilities. This leads to an initial set of guidelines for discriminating between platform-independent and platform-specific capabilities, described in terms of the "primary" data that contribute to the platform-independent "core algorithm", and the "supplemental" data that merely enhance, or "supplement", the core algorithm to exploit platform-specific capabilities. A corresponding methodology, described as a set of detailed design requirements, is proposed for the "modular treatment" of these supplemental effects relative to a stable, reusable core algorithm.

Related work in the robotics and software engineering communities is reviewed in Chapter 3, which leads to the identification of two detailed technical approaches that satisfy these requirements, described in Chapter 4. These techniques are applied to existing software for autonomous driving in Chapter 5, and the resulting artifacts are subjected to two established software analysis techniques in Chapters 6 and 7, informing the questions of "technical feasibility" and "design tradeoffs" listed above. Several candidate extensions of the autonomous driving components to novel input data are discussed in Chapter 8, which identifies several "likely points of variation" that can be expected in other robotic algorithms.

The second case study, presented in Chapter 9, reinforces the applicability of these ideas by looking into the complementary domain of planetary exploration. Similar problems of platform-specific adaptation are readily identified in a set of software components from CLARAty[39], and alternate designs are proposed that would enhance their adaptability in the same manner as for the more detailed case study on autonomous urban driving. The critical contributions of this work are then summarized in Chapter 10, which includes answers to each of the above questions, allowing practitioners to make well-informed decisions as to where, when and how they may apply the insights from this thesis to enhance the adaptability, and thus the reusability, of their own robotic algorithms.

Chapter 2

The Problem: Reuse Means Adaptation

One of the fundamental difficulties in reusing advanced robotic algorithms is the adaptation of those algorithms to make use of the specific set of data available on the target platform. This chapter examines the nature of this problem and distills the most basic units of change that compel this kind of adaptation, deriving a set of requirements for the modular treatment of this phenomenon.

2.1 Example: What Constitutes a Point?

Consider the common example of a "point cloud", as used by a wide variety of mapping, localization, terrain analysis, and other advanced robotic algorithms, and which may be derived using one or more of LADAR, stereo vision, or other perception techniques. Classical approaches promote reusability through highly generic input representations, as typified by the Cartesian coordinates in the `Point` class in Figure 2.1. As long as the contents of this class are representative of the information that a given robot can provide, then components such as the `TerrainAnalysis` class can be deployed on that robot without modification.

The trouble with robots, as alluded to in Chapter 1, is that while this representation covers the minimum data that are *necessary* to describe an individual point, it also excludes any *additional* data that a robot might provide *about* that point that could influence the `TerrainAnalysis` algorithm. For example, some LADAR scanners and stereo vision techniques can provide uncertainty information in addition to the standard Cartesian coordinates of a point. Accommodating this new information would require introduction of an

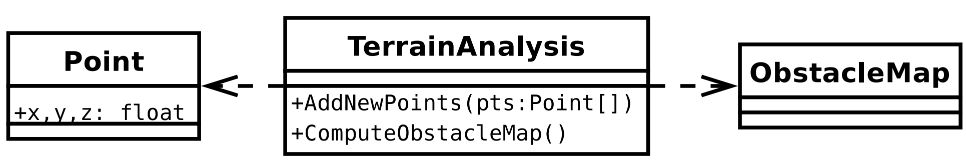


Figure 2.1: Reusability through generic data representations. The `TerrainAnalysis` class uses the minimal `Point` representation to generate an `ObstacleMap`. Notation: UML[48]

"error" member into the input `Point` representation, and extension of the `TerrainAnalysis` class to incorporate "error" in its internal calculations.

While this may seem trivial for any one such datum, subsequent robots may not be able to provide such "error" information, but may provide other data instead, such as temperature readings from stereoscopic thermal imagers, or an indication of the presence of vegetation through more sophisticated analysis techniques[8]. These data could also be relevant to the `TerrainAnalysis` algorithm, but they are not semantically compatible with "error" data, yielding the *semantic mismatch* shown in Figure 2.2.

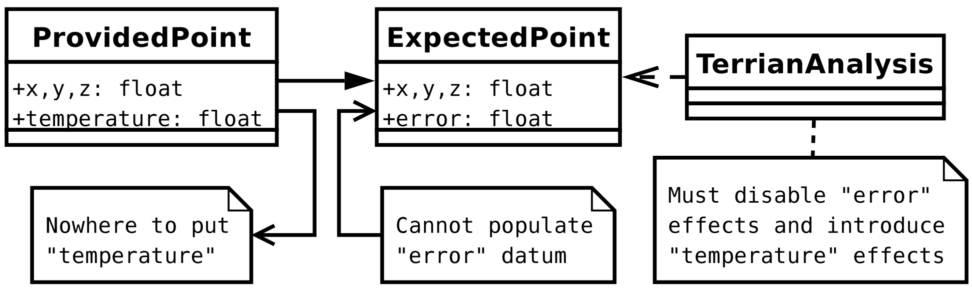


Figure 2.2: Semantic mismatch between an input representation that expects points annotated with error information, and a platform that provides temperature readings instead.

This mismatch can only be addressed by further modification of the `TerrainAnalysis` algorithm, such as to introduce mission-specific policies for whether or not "hot" surfaces or blocks of "vegetation" may constitute an obstacle. Moreover, these data do not generally have safe "default" values, so these policies would have to be selectively enabled according to the detailed capabilities of each specific robot. Especially when considering the possibility of interaction among these effects, such as for systems that provide both "error" and "vegetation" information, this can lead to a cumbersomely large set of conditionally-active effects that would significantly degrade the understandability and adaptability of the `TerrainAnalysis` component. This phenomenon is often described as an an accumulating "calcification" or "brittleness"[13] of robotic software with respect to this type of adaptation, and it is particularly common in prototype robotics, wherein individual sensors or perception algorithms are continually introduced or enhanced. In attempting to deal with this problem in a general way, typical approaches tend toward one of two extremes:

1. Attempt to accommodate all possibly-relevant input data directly within one "monolithic" component, or

2. Subdivide the component into modules that are small enough to permit the accommodation of any conceivable change through the substitution of modules or the rearrangement of their dependencies.

The "monolithic" approach emphasizes reuse of large software components at the cost of an ever-growing collection of dependencies on platform-specific data, the breaking of module opacity when adapting to novel inputs, and a high risk of semantic conflicts as described above. This pattern is quite common in robotic software developed using procedural or object-oriented languages, where the perceived "harmlessness" of "just one more configuration parameter" or "just one more input flag" leads to exactly the kind of brittleness described above. A common symptom of this is large blocks of commented-out or conditionally-compiled[1] code that is retained in case it can be used on the "next" system.

At the other end of the spectrum, a component can be subdivided into functional modules that are small enough that any conceivable adaptation can be implemented via the introduction of new modules and the rearrangement of the dependencies between them. In the limit, this hyper-modular approach leads to modules that are so small that the difficulties of re-use shift from the modification of the modules to the specification of their interactions. That is, the alteration of the component's configuration becomes at least as complicated and error-prone as directly editing the source code of the monolithic representation, leading to the same kind of "brittleness" with respect to further adaptation.

This pattern is commonly encountered in so-called "model-based" environments wherein all development is done through box-and-line visualization, where the boxes represent mathematical functions and the lines represent primitive data types, such as integers or floating-point numbers. While powerful in many respects, this representation is often criticized for its ability to take relatively simple mathematical models and spread them out over several pages of such boxes, each representing trivial mathematical operators such as addition, multiplication, etc., which are interconnected by a dizzying array of lines that represent various intermediate results. While this allows the modular accommodation of arbitrary input data without editing "source code", this comes at the cost of a set of interdependencies that is just as difficult to understand and adapt.

In either case, the implicit need to treat all possibly relevant input data leads to an *over-generalized* artifact (or collection of artifacts) that is extremely brittle with respect to further adaptation or reuse. While no practical system is likely to exist exclusively at either extreme, any attempt to build generically reusable robotic software components must somehow reconcile these opposing trends. The strengths and weaknesses of each extreme, combined with the unbounded span of possibly-relevant data, suggest that robotic algorithms cannot be fully generalized against any one set of input data, and that the interface that describes the data that an algorithm uses "as-is" must be complemented with an interface that describes the ways that an algorithm can be easily modified to accommodate more, less or different data in the future. Such an interface allows the algorithm to be more easily adapted to, and thus more easily reused across, many robotic platforms without the problems of accumulating "brittleness" described above.

2.2 Compelling Adaptation

This accumulating "brittleness" was particularly apparent during the development of the autonomous driving software discussed in Chapter 5, and it was one of the primary motivators of this work. However, the problems of adaptation to varying input data are not unique to prototype robotics, nor do they require explicit "porting" from one platform to another as in the example above.

In a more rigorous production environment, such as the aerospace or automotive industries, similar difficulties may also be encountered in the use of the Product Line[6] approach to maintain autonomous and semi-autonomous driving algorithms across a collection of re-

[1] Such as by `#ifdef MY_ROBOT ... #endif` in C/C++.

lated, but distinct vehicles. In this case, the goal of sharing components across the entire line can be thwarted by the differences in sensing and actuation capabilities that separate one class of from another, such as a compact sedan vs. a luxury SUV. For example, some platforms in a product line may detect traffic using RADAR, where others might use computer vision techniques instead. For an advanced algorithm, such as a highway merge planner, to be reusable on all such platforms, it must be implemented in terms of the commonality between these techniques, such as providing basic position or range information about candidate obstacles. For optimum performance on any one system, however, the algorithm must also be adaptable to the specific strengths of each sensing technique, such as exploiting higher confidence in velocity measurements from RADAR, or the ability to detect geometry and turn signals with computer vision.

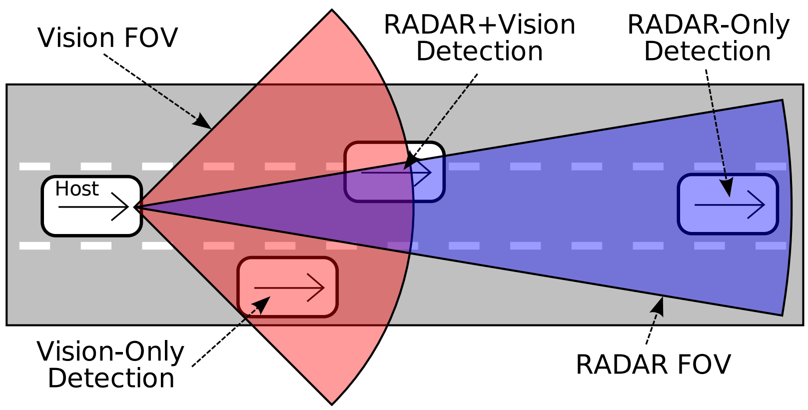


Figure 2.3: Differences in sensor ranges and coverage patterns can compel run-time adaptation to sensor-specific data.

In the case of multi-modal perception techniques, such as the use of both RADAR and machine vision on one platform, the design-time problem of accommodating platform variability becomes a problem of run-time adaptation to sensor-specific data. As illustrated in Figure 2.3, differences in sensor range and coverage patterns can cause obstacles to be detected by any combination of sensors, requiring algorithms to selectively enable individual sensor-specific effects according to the contributors to the detection of any one vehicle. In a sense, this compels a type of runtime adaptation in autonomous driving algorithms to exploit specific capabilities when available and otherwise degrade to the "best possible" functionality when not.

Taken one at a time, these issues can typically be reduced to the introduction and inspection of a single status flag, such as "using a vision sensor", and triggering one of two sets of functionality depending on the result. In isolation, this may seem like a trivial problem, but the number of such effects that can accumulate in trying to support multiple platforms, or even multiple sensing modalities on a single platform, can lead to a cumbersomely large set of these "simple" flags to maintain. This is compounded by the possibility of interaction among multiple effects, such as deriving special cases for "using a vision sensor", but also "using a RADAR sensor", that leads directly to the "brittleness" at the focus of this thesis.

2.3 Units of Adaptation

Regardless of the specific impetus, the need for adaptation arises because a *new* set of data is available that is different from some *previous* set of data that was expected by a given component. More specifically, the *new* set of data may have one or more:

1. **Additional** data to be incorporated to enhance performance, safety, etc.;

2. **Absent** data that can no longer be derived by the system, requiring deactivation or excision of any associated effects;

3. **Altered** data, whose precise semantics “drift” as a result of progressive enhancements to the surrounding system, requiring frequent updates to the corresponding effects.

Any technique that attempts to treat the effects of these changes in a modular way must be able to treat them on an individual basis, as the changes may come one at a time, such as during the active development of a single robotic system. Even if several changes occur all at once, such as when migrating a component to a completely new system, it is sufficient to model the consequent adaptations as a discrete sequence of changes for individually additional, absent or altered data. Thus, it is possible to analyze the implications of each type of change in isolation, and take the union of the requirements they impose as the set of requirements for any design technique to effectively treat this problem.

2.4 Implications of Adaptation

First and foremost, the adaptation of a given component to *additional* data implies that the component can already function effectively without those data. That is, the new data are capable of, or perhaps are specifically meant to, *supplement* the existing algorithm to enhance performance or yield more accurate results. A modular treatment requires the corresponding *supplemental effects* to be bound to the algorithm through a dedicated *adaptation interface* that minimizes invasive modifications to the original component. Relatedly, the use of this adaptation interface should not require intimate knowledge of the component’s inner workings, as the effort to understand the context and consequences of an adaptation is the primary cost to be reduced by avoiding such direct modifications.

Conversely, the adaptation of a given component in the face of *absent* data begins with the assumption that the component can function effectively without those data, but that its behavior will somehow be degraded as the effects of the absent data are removed. That is, the absent data are also *supplemental* to the algorithm, as described above. The corollary to this is that there is some set of *primary* data that are necessary to enable baseline or *core* functionality, and that all other possibly-relevant data can be treated as *supplemental*. A modular treatment requires the effects of the absent data to be completely and separately encapsulated such that those effects, along with any corresponding compile-, link- or load-time dependencies on the supplemental data in question, can be easily removed by excluding the appropriate module(s). As with the treatment of *additional* data, this exclusion should require minimal, if any, invasive modifications to the *core* component.

Lastly, the adaptation of a given component relative to *altered* data implies that the data can actually have several specific meanings. This generally excludes data that can be taken entirely at face value, such as locations, sizes, or anything else that represents a concrete measurement in real units, leaving data that have much deeper semantics, such as inferred error measurements, or enumerations that represent some abstract state of the environment, such as the inference of driver intent through observed behavior. These typically represent some of the most subtle facets of the confluence of sensors, algorithms and application domain that are unique to a given platform, and their specific meanings are the most likely to evolve as the platform evolves. The alteration of the semantics of such data implies that the effects of that data must be revisited and synchronized with the new meaning. The modular treatment of this synchronization requires that the effects of the altered data be co-located and able to be examined and modified in isolation without direct modification or significant knowledge of the rest of the component under adaptation. This can also require the introduction of completely new effects, such as with *additional* data, and the excision of invalid effects, as with *absent* data, further reinforcing the implications of both.

2.5 Requirements for Adaptation

Distilling the above yields the following requirements for any design that attempts to separate the reusable elements of a given *core algorithm* from the individual *supplemental effects* that would bind that *core algorithm* to any one robotic platform. In order for the resulting software artifact to be easily adaptable to present and future platform-specific *supplemental data*, a candidate design must:

AR.1 Differentiate the *primary* data that enables an algorithm's baseline functionality from *supplemental* data that merely enhances that algorithm to exploit platform- and/or context-specific information.

AR.2 Identify a unit of encapsulation for the treatment of the *core* algorithm as a strict function of the *primary* data.

AR.3 Provide a dedicated interface that allows the behavior of the *core* algorithm to be adapted to a wide variety of possible *supplemental effects* with minimal modification to its implementation and without knowledge of its inner workings.

AR.4 Provide a unit of encapsulation that allows the effects of arbitrary *supplemental* data to be bound to the aforementioned interface such that those effects can be introduced, removed, and updated by means of modular augmentation, exclusion and substitution.

These requirements represent the core methodology proposed by this thesis, which is aimed at enhancing the adaptability of advanced robotic software relative to *additional*, *absent*, or *altered* data, as discussed above. Many detailed design techniques can be used to fulfill these requirements, with the most appropriate one depending on details such as programming language, team organization, and individual developer experience. To put these details in an appropriate context, and to frame the selection of the specific techniques that will be used later in this work, the next chapter reviews existing approaches to reusing robotic software and examines related work from the software engineering community that addresses similar issues of adaptation and reuse.

Chapter 3

Related Work

The topic of designing reusable and adaptable software components has been of great interest to both the robotics and software engineering communities for some time, and the resulting space of techniques for adapting software to new environments is known to be vast and various[29]. Rather than presenting a dense and disorienting attempt at an exhaustive survey of the field, this chapter focuses on work that specifically relates to or otherwise motivates the problem of adaptation to changes in the data provided to robotic algorithms. Section 3.1 begins with a discussion of the state of reusable robotic software, focusing on several commonly-encountered examples that typify the broader scope of techniques currently in use in the robotics community. Thereafter, Section 3.2 summarizes the traditional notions of adaptation and reuse in more general software engineering, and presents the emergent techniques of Aspect-Oriented Programming and Crosscutting Programming Interfaces.

3.1 Reuse and Extension in Robotic Software

The promotion of reuse and adaptation in robotics has, to date, taken one of four basic forms: specifications, application frameworks, domain-specific languages, and "behavior-based" architectures. Many examples of each exist; a few are described here in terms of what they aim to provide, and how they relate to the reusability of robotic software components.

Specifications

One of the most direct methods of promoting reusability in robotics is to develop standards and specifications for reusable components. One such emergent example is the Joint Architecture for Unmanned Systems (JAUS)[27]. JAUS specifies the bit-level message definitions, interface protocols and expected behaviors of several functional robotic modules in a service-oriented architecture. System components such as mobility platforms, position sensors and manipulator arms are modeled as software services, each providing an interface to the associated hardware device according to the JAUS specification for that class of device.

While this is an excellent step for promoting reuse and interoperability among robotic hardware components, JAUS currently falls short in specifying higher-order system elements that would couple the functionality of these devices to perform complex tasks. Moreover, in attempting to define generic messages that can fully capture the various components that can comprise a JAUS-compliant system, the specification implicitly places the burden of interpretation on either a human operator[1], or on an otherwise custom client application that binds specific sensor data to specific device actions to accomplish some specific mission. The reusability of such "custom clients" of JAUS and similar standards-compliant hardware interfaces is the focus of this thesis, as these client applications must somehow accommodate a wide variety of sensors, actuators, and modeling techniques.

Frameworks and Toolkits

At least within the robotics research community, and to a significant extent in robotic industry, application frameworks and their associated "toolkits" of components are the dominant means of reusing advanced robotic software components. As discussed in Chapter 1, these frameworks provide important support for rapid prototyping of new technologies, but they often constrain developers to a narrow set of supported devices, and reuse of components outside of these supported platforms presents significant and ongoing challenges. These are alluded to in anecdotal "lessons learned" in the application of component-based architectures to robotic systems, which point to the difficulty balancing generality and specificity in robotic software, especially in architectural case studies or summary presentations of individual frameworks or toolkits.

CARMEN[37] and Player[11] are two popular examples of these toolkits, providing components for robot control, localization, mapping and navigation, along with simulation and visualization capabilities. Reuse of the software provided with these toolkits is straightforward as long as the test platforms are very similar to those supported by the toolkit, but their deployment on platforms with novel sensors, and interoperation with components from other toolkits remain challenging problems. For instance, many of the mapping and localization algorithms in CARMEN depend on the use of a narrow class of planar laser scanners, such as the Sick LMS-120, to function correctly. While this implicit need to procure a specific laser is generally accepted in the research domain, it is a classic example of how comparatively abstract software components can depend on relatively "low-level" platform details.

The dependency on precise data representation and semantics is further highlighted by the MARIE project[12], whose goal is to allow components from multiple toolkits to interoperate on a single system. MARIE introduces a "Mediator Interoperability Layer" (MIL) that converts the message data between a given component's native representation and a highly generic intermediate format. The generation of the so-called Communication Adapters (CA's), that perform the conversion to and from this generic data representation is acknowledged as a critical and difficult activity that depends heavily on the level of "mismatch" between the data that is provided and the data that is expected. This is described both in terms of "architectural mismatch"[23], which refers to conflicts between, for example, polling- vs. notification-based data delivery methods, and also in terms of

[1] The original JAUS specification was focused heavily on teleoperation of small mobile platforms and manipulator arms.

"semantic mismatch", which refers to the difficulty of achieving an accurate translation between "similar", but not "identical" inputs, as highlighted by the differences between `ExpectedPoint` and `ProvidedPoint` in 2.2.

This difficulty of effectively representing variations on similar data is also explored in the work surrounding CLARAty[38], the goal of which is to provide a reusable software framework and collection of reusable tools for NASA's planetary rover program. A discussion of the challenges[39] faced in developing this framework lends tremendous insight into the barriers to reuse and adaptation that are imposed by the nature of evolving and heterogeneous robotic platforms. In particular, many of the challenges encountered during the development and use of CLARAty "stem[med] from the variability of robotic mechanisms, sensor configurations, and hardware control architectures"[39]. When attempting to determine an effective abstraction for a given class of devices, such as motor controllers, neither the union of all possible capabilities, nor their intersection were deemed satisfactory. The former would lead to over-generalized interfaces that are cumbersome to use, and the latter over-simplifies the interface, lacking the ability to represent any special properties of the underlying platform which may be critical to the effective operation of the robot. The difficulty and importance of achieving a balance between these extremes is highlighted, which is consistent with the problem of adaptation to changes in input data that is explored by this thesis. This similarity led directly to the inclusion of CLARAty components in a complementary case study, presented in Chapter 9, which demonstrates the applicability of the proposed methodology beyond the scope of autonomous urban driving.

Behavioral Abstractions and Domain-Specific Languages

Beyond specifications and frameworks, there has also been a great deal of work in the robotics community that examines design techniques related to reactive behaviors, task description and mission execution. These typically provide interesting insight into and important support for some specific requirements of robotic systems, but generally represent matters that are orthogonal to the types of algorithms studied in this thesis.

For example, the Task Description Language[50] provides a set of C++ language extensions for task decomposition, synchronization and monitoring that reduces the difficulty of decomposing robotic tasks into a collection of individual sub-tasks. TDL has been used effectively on several systems, including within the CLARAty system mentioned above[17]. Its fundamental goal is to provide representation and support for breaking tasks into recursively smaller tasks, where the algorithms that perform the tasks, such as waypoint-based navigation algorithms, are viewed as external dependencies. Similarly, finite-state-machine representations of robot behavioral systems[2], while popular and very useful for modeling the behavior of complex systems, do not directly consider the algorithms that trigger the transitions between various states. For instance, the transition in an autonomous vehicle from an abstract state "Waiting at an Intersection" to another abstract state "Driving Down a Road" must be governed by some software component that reasons about the available data to determine whether it is the robot's turn at the intersection. Such algorithms are clearly subject to the types of adaptation as laid out in Chapter 2, and are thus within the scope of this thesis, as opposed to any particular arrangements of states and transitions, which are orthogonal to the issues explored in this work.

One notable exception to this orthogonality is Brooks' subsumption architecture[9], which allows the alteration of the behavior of a component by introducing new modules that can conditionally *suppress*, or replace the output, of an existing module. This mechanism mirrors some of the representational power of the software engineering techniques discussed in the next section, but is focused on adapting the data flow between individual modules in a pipe-and-filter decomposition. One key criticism of this approach is that it leads to a hyper-modular decomposition if applied vigorously as the primary means of modularization, leading to a complex tangle of interdependencies that are as intractable as a monolithic encoding in a single module.

3.2 Software Engineering

Looking beyond the scope of robotics, the wider software engineering community has been concerned with the reuse and adaptation of software from its earliest beginnings. In a sense, nearly all research in software engineering touches on the desire for reusable software components, and many useful techniques have been developed that address many of the challenges of reusing software across multiple systems.

In particular, the idea of complementing a component's "traditional" functional interface with a dedicated configuration or adaptation interface can be traced at least as far back as 1996, with the introduction of Open Implementations[31]. This work, which is viewed as a precursor to the Aspect-Oriented methodology described in Section 3.2, recognizes the problem that:

> It is impossible to hide all implementation issues behind a module interface. Some of these issues are crucial implementation-strategy decisions that will inevitably bias the performance of the resulting implementation. Module implementations must somehow be opened up to allow clients control over these issues as well.[31]

These "implementation-strategy decisions" reference the many tradeoffs that must often be made between code size, execution speed, memory consumption, and other such "non-functional requirements"[6] of a software system. In order to realize a given software module, developers must assume an application context and choose among such tradeoffs according to an expected usage pattern. Thereafter, those choices would be hidden behind the module's functional interface, leaving clients of that interface with little recourse if their application conflicts with the original developers' assumptions.

As an example, the developer of a windowing system that provides functionality for drawing text, capturing mouse input, etc., may make such non-functional tradeoffs assuming a typical application will have relatively few windows, perhaps favoring execution speed over memory consumption. If a client of the windowing interface were to instead attempt to create thousands of small windows, such as for a spreadsheet application, then the memory performance of the windowing system may be completely unacceptable.

To address these and similar issues, the authors of Open Implementations propose a kind of "complementary" interface that provides clients with a degree of control over issues that are otherwise hidden by a module's "traditional" interface. In the case of the windowing example above, the authors suggest the use of preprocessor (i.e., `#pragma`) directives, to allow clients to select between memory- and speed-optimized "back ends" to the same functional API. Although this does not directly consider the effects of novel input data, which are the focus of this thesis, the underlying problem of having to alter otherwise "hidden" functionality in order to fully (or even successfully) integrate a component into a larger system, resonates with the issues outlined in Chapter 2, and is a recurring theme in other areas of research into software design.

Encapsulation and Data Hiding

The principles and virtues of encapsulation and data hiding that came to be known as Object-Oriented (OO) design are well-recognized by the software engineering community[43]. This methodology promotes adaptation and reuse by identifying decisions or requirements that are likely to change and "encapsulating" the affected parts of a software system behind an abstract functional interface. In so doing, implementation details that are tied to volatile system requirements, such as the selection of alternate algorithms or data representations, can be varied without affecting clients of the abstract interface. Common patterns of OO decomposition, such as those presented in [21], have gained mainstream acceptance within the robotics community, and they are the principal method of adaptation and reuse in most of the frameworks and toolkits discussed above. These techniques have been used to great effect in many large-scale robotic systems, including the Tartan Racing software system[58], and more specifically within its Behavioral Executive subsystem[4], which is the focus of the experimental work discussed in Chapter 5. As such, a typical Object-Oriented approach that might be used separate a core algorithm from its supplemental effects is included in the experimental approach laid out in Chapter 4.

However, the widespread exploitation of the strengths of Object-Oriented techniques has been accompanied by an accumulating understanding of its limitations, particularly in the treatment of certain secondary design concerns that are conceptually coherent, such as transaction logging or the enforcement of access permissions, but must otherwise permeate an object-oriented decomposition of a system. Such *crosscutting concerns* are difficult to encapsulate in an object-oriented manner because their implementation must intrinsically be scattered across many methods of a single object, and often across many objects in a given software system. This has strong parallels to the possible effects of adding, removing or altering data as described in Chapter 2, which can be similarly scattered across the implementation of one or more algorithms in a robotic system. Aspect-Oriented methodology, discussed in the next section, provides a means of collecting various and otherwise disparate effects into a single unit of encapsulation, an *aspect*, providing interesting possibilities for binding supplemental effects to a given core algorithm.

Aspect-Oriented Programming

Aspect-Oriented Programming[33] provides a means of coherently and modularly describing a software concern that must otherwise, by necessity, be *scattered* across and *tangled* with many elements of a complex software system. It introduces the notion of separating a system into:

1. A *dominant decomposition* that uses traditional techniques to compartmentalize and subdivide the system's core functionality free of scattering or tangling concerns;

2. A collection of *aspects* of the system that describe those *crosscutting concerns* relative to the structure and execution of the dominant decomposition.

These two are combined to form a complete system by an *aspect weaver*, which is analogous to a traditional compiler for procedural or object-oriented languages. While the functionality of the aspect weaver can be seen as a source-code transformation tool, it is important to note that the AO model is based on the interception of execution points in the underlying system, which are, by their nature, a runtime phenomenon. These execution points, called *join-points*, are described in terms of the implementation of the dominant decomposition by specifying class names and method and member signatures in so-called *pointcut* declarations.

Most mature aspect languages and compilers, such as AspectJ[32], Hyper/J[42], and AspectC++[19] support wild-card characters in the declaration of a pointcut, allowing the specification of large sets of join-points in a compact form. For example, the pointcut "`calls( "void MyClass::Set%(int)" )`" specifies all calls to methods of `MyClass` that begin with the letters "`Set`" and take a single integer as a parameter. A subsequent *advice* declaration then specifies functionality to introduce at each matching join-point, which can either extend or override the functionality of the underlying system as the developer sees fit.

Within the domain of complex software systems, it has been shown [22] that the use of aspect-oriented techniques can dramatically increase the conceptual coherency of many crosscutting concerns in otherwise common object-oriented design patterns[21], especially promoting such desirable qualities as reusability, maintainability and adaptability. As an example, Appendix B demonstrates the use of AO techniques on a simple C++ class to encapsulate the crosscutting concern of maintaining thread-safe access to that class.

Crosscutting Programming Interfaces

There are, however, two possible drawbacks to the application of aspect-oriented techniques to a software system:

1. In specifying pointcuts relative to specific class, method, and variable names, aspects acquire a brittle dependency on the implementation details of the underlying system.

2. In being able to override arbitrary functionality in the underlying system, it is difficult to make guarantees about performance, correctness, etc. of an aspect-woven system.

The former phenomenon is often called dependency *reversion*, as the underlying system is no longer subject to the scattering and tangling of crosscutting concerns, but, in exchange, the aspects that implement those concerns are highly dependent on the detailed implementation of the underlying system. This is widely recognized as one of the most significant challenges faced by an aspect-oriented developer, especially while the "underlying" system is still under active development. Subtle alterations to the method, member and class names can break the binding of crosscutting concerns to the underlying system, yielding erroneous, and typically mysterious, results.

The second problem, of the invasive power of aspects, and the corresponding vulnerability of underlying components, has been explored in many contexts in the AO community, particularly in attempting to classify aspects according to how drastically they affect the underlying system[28]. These recognize that the aspect-oriented techniques may allow too much freedom of interaction with the underlying system, and that some means of policy checking and enforcement may be necessary to be able to make broad guarantees of performance, correctness, etc.

Researchers have recently proposed the use of "Open Modules"[1], and later "Crosscutting Programming Interfaces" (XPI's)[24] to address this necessity. While the former is slightly more focused on the theoretical formulation, and the latter on the practical implementation, they each describe the need to:

1. Isolate aspects of a given "core" component from having to delve into, and become directly dependent on, the details of the component's implementation to find appropriate method and member names for the application of advice.

2. Provide the maintainers of the "core" component with some flexibility in renaming and refactoring elements in the component without interfering with potentially numerous and unknown aspects that depend on those details.

3. Allow some way to limit the scope of aspect-oriented modifications in order to be able to make, or at least construct, significant guarantees about the module's functionality and performance under the influent of aspects.

In both cases, the proposed solution is to complement the core module's functional interface with a set of named pointcuts that represent the ways that an aspect is *allowed* to modify the core module's behavior.aOpen Implementations?RESOLVED: extended the prose immediately hereafter to fold the idea of Open Implementations back in. JA:This might be connected more strongly to Open Implementations. In many ways, these can be seen as more general, and more powerful, variations on the theme of "Open Implementations", discussed above, in that they allow the underlying *behavior* of an algorithm to be modified through a non-standard interface. The critical benefits, which parallel the benefits gained from traditional OO encapsulation, are that the source-level details of the core module are hidden from from client aspects, and that client aspects are restricted to augmenting a core component only as permitted by its crosscutting interface. These properties are highly desirable when considering the *primary* vs. *supplemental* treatment of robotic algorithms outlined by this thesis, so the second detailed design technique outlined in Chapter 4 uses an XPI to encode the "likely points of variability" in a given algorithm, applying individual supplemental effects using AO *advice* directives.

Supplemental Effects and Aspect Classification

Beyond issues intrinsic to AO methodology, it is also important to consider that the effects of supplemental data are highly algorithm-specific and are thus not fully "crosscutting" in the traditional sense of the term. While this work will show that there are several common *categories* of supplemental effects in robotic systems, it is important to note that each *individual* supplemental effect represents a unique combination of data semantics and application context that will not be broadly applicable to multiple components, or even to multiple places in a single component. In addition, these effects are explicitly meant to alter the functionality of the core algorithm to include additional information and yield improved results, which is contrary to typical examples from the AO literature. These focus either on adding benign side-effects, such as the canonical example of debug tracing, or else temporarily suspending or diverting existing functionality while leaving it otherwise unchanged, such as in other common examples of maintaining thread-safety or certain authentication and security concerns.

Recent work on aspect analysis and classification[28] labels these more typical examples as *spectative* and *regulative* aspects, respectively, and identifies two additional categories of *invasive* aspects that alter the computational flow (and results) of the underlying algorithm. The first such category, of *weakly invasive* aspects, is defined as modifications whereby state transitions in the aspect-augmented system "...begin [and end] in states that already existed in the state graph of the underlying system (perhaps for different inputs from those in the augmented system)."[28]

As an example, the authors suggest an aspect that imposes a discount policy for prices in a transaction system, where the ultimate system outputs would be the same as if prices were set lower to begin with. In terms of the simplified example from Figure 2.1, an augmented `TerrainAnalysis` class would still produce the `ObstacleMap` output, but with different specific values depending on the nature of the augmentation. This, along with the broader category of weakly invasive aspects, resonates very strongly with issues of adaptability in robotic systems, where the special capabilities of a given robot are often used to enhance an algorithm's performance in ways that remain hidden behind its output specification. This is especially true of layered or "hybrid" control architectures such as in [36, 58], wherein each layer embodies a complex reasoning algorithm that could be enhanced by any number of robot-specific capabilities. As long as these enhancements remain *weakly invasive*, they would make good candidates for *supplemental* treatment as described in Chapter 2.

Discriminating between these and more *strongly invasive* enhancements remains an open challenge that is highly analogous to the problem of deciding where to draw the algorithm-specific division between core and supplemental effects as specified in Requirement **AR.1**. Ultimately, the designer's intuition must be applied to determine the line between the two, and the primary goal of the experimental work discussed in the following chapters is to inform future designers of the costs, benefits, and pitfalls of the primary vs. supplemental methodology at the heart of this thesis, enabling them to make effective decisions as to when and how to apply it to their own problems.

Chapter 4

Technical Approach

This chapter presents the technical approaches that will be applied in this thesis to the discrimination of primary vs. supplemental data, and the isolation of the corresponding core algorithms from their supplemental effects. Section 4.1 begins with the division between primary and supplemental data, focusing on *descriptive* rules that can be applied to an existing software system, where more *prescriptive* guidelines for use when designing new systems are among the central contributions presented in Chapter 10. After describing the basic criteria for identifying supplemental data and effects, two detailed software design patterns, one Object-Oriented and one Aspect-Oriented, are presented in Sections 4.2 and 4.3, respectively. Section 4.4 outlines the experimental application and analysis of these two design techniques in the context of existing software for autonomous driving behaviors, along with a complementary case study that evaluates these techniques in the domain of planetary exploration.

4.1 Primary vs. Supplemental Data in Existing Systems

As highlighted by Requirement **AR.1**, identifying an appropriate division between primary and supplemental data is critically important to the successful application of the methodology proposed in this thesis. Too much emphasis on *primary* data migrates the design toward a "monolithic" representation, with the corresponding problems of "black box" adaptability discussed in previous chapters. Conversely, too small a set of *primary* data may reduce the core algorithm to the point of uselessness in isolation, forcing the treatment of practically all data as *supplemental* and yielding an equally undesirable "hyper-modular" design. While this stresses the importance of an appropriate division between the two, it also highlights one of the strengths of the proposed methodology: that moving individual data "back and forth" across the line between *primary* and *supplemental* allows an incremental exploration of the space between monolithic and hyper-modular designs.

Whether a given datum should be treated as *primary* or *supplemental* is highly dependent on the algorithm in question and the degree of adaptability sought by the designer. Drawing on the point-cloud example in Chapter 2, an indication of whether or not a point is "vegetative" is clearly *supplemental* to the generation of an obstacle map. That is, a valid obstacle map can be generated in the absence of "vegetation" data, but the presence of such data can be used to enhance, or *supplement* the obstacle map to, for instance, cause the robot to avoid areas of "dense vegetation", if possible. However, this could easily be *primary* for other algorithms in the same system, such as for selecting navigation goals that seek out areas of dense (or any) vegetation for scientific analysis.

The inherent subjectivity of these matters makes it difficult, if not impossible, to derive a single, objective classification scheme for *primary* vs. *supplemental* data. Still, it is possible to provide softer "guidelines", rooted in specific examples, to inform the judgment of future designers in classifying a given datum according to:

1. The intrinsic nature of the datum in question, such as whether it can be taken at face value, whether it has a meaningful "default" value, and how likely it is to be available on other robots.

2. The nature of the effect that the datum has on a given algorithm, such as whether it induces isolated, minor behavioral changes, or instead compels a fundamental shift in the approach to the problem.

The first category, of intrinsic properties of candidate supplemental data, is discussed here in detail, and these guidelines will be used to seed the experimental work at the core of this thesis. One main goal of this experimental work is to provide insights into the latter category, the types of effects that supplemental data can have on robotic algorithms, which are among the critical contributions of this thesis. As such, the second category is discussed only briefly here, with more in-depth recommendations discussed throughout refactoring and analysis in Chapters 5-9, and summarized in the concluding Chapter 10.

The Nature of the Data

When trying to determine whether a given datum should be treated as *primary* or *supplemental* for a given algorithm, the most straightforward question to ask is whether will be available on the "next" robot or on a "typical" robot in "similar" application domain. If so, then the datum is a strong candidate for *primary* treatment, as it is not particularly likely to induce the types of adaptation described in Chapter 2. Otherwise, it is a strong candidate for *supplemental* treatment, as it will be much more likely to compel those types of adaptation.

In terms of existing algorithms that are already deployed on multiple platforms, or when designing algorithms to be shared across a particular Product Line[6], a simple set-theoretic view, as shown in Figure 4.1, can be used to describe the set of data that *can*, but not necessarily *should*, be treated as *primary* data.

At first glance, it may seem counter-intuitive that the set of *primary* data might deviate from the intersection of known-platform capabilities, but there are two critical factors that a designer must consider that lead to the consideration the intersection as more of a "soft upper bound" than a direct description of the set of primary data. First and foremost, there is an important duality to this problem in that while the intersection of current platform capabilities can certainly be used to *describe* a set of *primary* data, the set of *primary* data for a given algorithm will also *prescribe* the set of supported platforms for that algorithm. That is, any data within the current intersection that a designer can frame as *supplemental* will effectively increase the scope of possible future platforms that will be supported by that algorithm. The critical tradeoff to this is the expected difficulty of treating the excluded

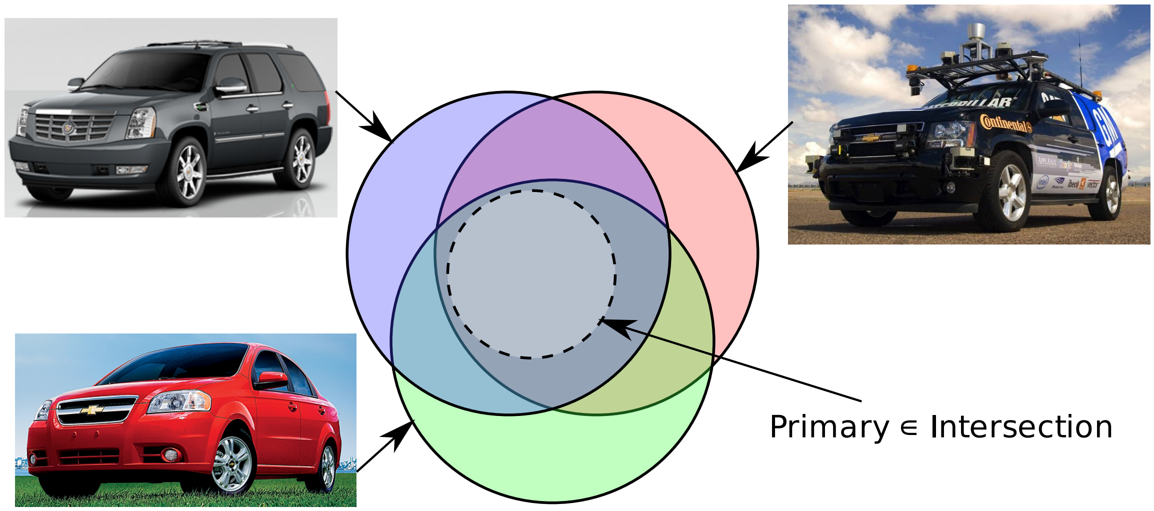


Figure 4.1: Set-theoretic determination of primary vs. supplemental data: the set of primary data is bounded by the common capabilities of all known platforms.

data as *supplemental*, which depends on the nature of their effects on the algorithm in question and is discussed further in Section 4.1.

The potential difficulty of separating out *supplemental effects* raises the second critical factor that must be considered by a designer when identifying an algorithm's *primary* data. Namely, the intersection of known-platform capabilities may be too restrictive to allow for the development of single shared *core* algorithm[1]. It is also possible that some subset of capabilities outside the present intersection may compel a substantially different approach to the problem at hand. In either case, the designer may choose to designate data outside of the current intersection as *primary*, effectively reducing the set of platforms to be supported by the original algorithm and requiring the development of an independent solution for the de-scoped systems. The tradeoffs are once again tied to nature of a datum's effect on a given algorithm, except that in this case the "benefit" is less about supporting multiple platforms than it is about simply acknowledging that no one *core* algorithm can effectively support *all* desired platforms. While undesirable, this is a choice often faced by designers in trying to factor out common components: whether or not the underlying functionality is "common enough" to warrant extraction and sharing, or whether there would be too much overhead (code size, efficiency issues, etc.) to make it worthwhile..

An interesting confluence of these two themes occurs frequently during the development of research prototype robots, where many data that are expected be available later in a project may not be available in the "current" version of the system, and the viable capabilities of the "final" system are not known *a priori*. In this case, treating initially-unavailable data as *supplemental* would support early system testing, and would also provide a certain amount of risk mitigation by building degraded modes of operation into the core algorithm. However, these benefits must be weighed against the expected costs of

[1]The limit of this case is the empty intersection, where there is no "common denominator" among platforms to exploit for a shared core algorithm.

maintaining separate *supplemental effects*, especially given the possibility of unknown future capabilities that may compel fundamentally different approaches to the problem. In either case, a central goal of this thesis is to inform the designer's judgment as to the nature of "typical" supplemental effects so as to help him identify when and how an algorithm's set of *primary* data would deviate from the pure intersection of known or expected platform capabilities.

Beyond simple set theory, it is also important to consider the *semantic depth* of a given datum, or how strongly its meaning is open to (or requires) interpretation. If it can be taken at face value, such as a measurement in real units, then it is a strong candidate for *primary* treatment, as alternate representations are typically limited to unit or coordinate-frame conversions. Moreover, such data are more likely to have meaningful notions of "default" or "nominal" values, so adapting a component to their absence may be a simple matter of assigning these "default" values. For an example from the autonomous driving software discussed in Chapter 5, when the "complete" geometry of a vehicle could not be detected, the system assumed a "default" size of 3m by 5m, and fitted the detected facets of the obstacle to this "nominal" geometry. This proved to be a safe assumption in all contexts, so even though the actual size of a detected vehicle could not be modeled accurately at all times, the *size* data remained semantically stable, and did not compel any of the types of adaptation laid out in Chapter 2.

It is important to note, however, that there are more subtle semantics bound to this data than simply the length and width of the detected vehicle. That is, the idea of fitting that geometry to the "visible facets" embeds an assumption about sensing and modeling techniques, and the "default" geometry embeds an assumption from a particular application context, such as "typical" suburban traffic, that may be invalid in another application, such as at a construction site. Although these particular assumptions are fairly weak, and their violation would no likely lead to critical failures, other data may embed more significant assumptions about application context that point more strongly toward *supplemental* treatment.

The Nature of the Effect

To discriminate primary vs. supplemental for such a datum with more significant *semantic depth*, a developer must also consider the nature of that datum's effect on the algorithm in question. As discussed at the end of Section 3.2, this issue is highly analogous to the discrimination of *weakly* vs. *strongly* invasive aspects, which remains an open and interesting challenge. That is, data with weakly invasive effects, especially those that remain hidden behind the algorithm's existing output specification, will be better candidates for supplemental treatment than data that have a more strongly invasive effect on the core algorithm.

As with the issue of "volatility" discussed above, identifying the "degree" of invasion is ultimately left to the judgement of the designer, and one of the goals of this thesis is to inform that judgement through a catalog and categorization of supplemental effects in existing software components. Understanding how platform-specific data affected these components will help future designers identify and forecast similar issues in their own systems. When coupled with knowledge of the tradeoffs to expect among multiple technical approaches to the problem, as provided by the alternate designs and analytical techniques discussed below, this will allow future designers to make informed decisions as to when and how to apply the proposed primary vs. supplemental methodology to their own software design problems.

4.2 Object-Oriented Technique: Delegation

With a distinction between *primary* and *supplemental* data in mind, it is now necessary to identify candidate design techniques that satisfy Requirements **AR.2** through **AR.4** from Chapter 2. As a running example for discussion of these detailed design techniques, consider the following problem of determining whether or not it is “safe” to merge into moving traffic at a T-intersection, such as shown in Figure 4.2.

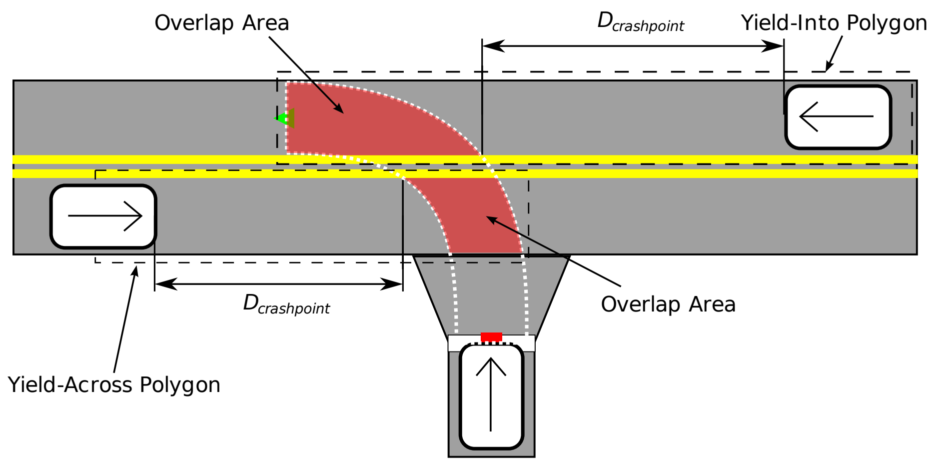


Figure 4.2: Example intersection yield scenario.

This problem, which will be revisited in more detail in Section 5.3, consists of:

1. Identifying, from a large set of candidate “moving obstacles”, those that are “relevant” to the yield calculations;

2. Determining, for each such “relevant” obstacle, the estimated distance and time to the “crash point” for the pending intersection maneuver;

3. Comparing those estimated “crash times” to the expected time necessary to complete the maneuver in order to determine whether there is “enough room” to proceed.

One of the particular uses of *supplemental* data in this algorithm was to cull probable “false positives” from the set of candidate obstacles. This was accomplished, in addition to testing for geometric overlap with the occupancy polygons illustrated in Figure 4.2, by requiring that a candidate obstacle:

- Be “observed-moving”, which implied historical motion consistent with “typical” Urban Challenge traffic (see effect **PE.S.3** in Ch. 5, p. 44, for more details), and

- Have at least one strong "lane association", which meant that its historical motion was also consistent with travel along an actual road lane, as opposed to more erratic or off-road driving behavior (see effect **PE.S.1** in Ch. 5, p. 44).

```
void PrecedenceEstimator::computeYields() {
 // set up yield calculation (~150 lines of code):
  // determine where traffic may come from
  // iterate over list of moving obstacles
  // test each for relevance...
  if(obstacleInYieldZone(obst) &&     // core concern
     obst->isObservedMoving &&        // supplemental effect
     !obst->laneAssocations.empty()) // supplemental effect
  {
    // main body of yield calculation: ~200 lines of code
  }
  // cleanup and set appropriate outputs: ~50 lines of code
}
```

Listing 4.1: Direct encoding of supplemental effects in the original Precedence Estimator.

As shown in Listing 4.1, these "extra" conditions were originally embedded in a large, complex method, which made them difficult to identify, understand, and update during ongoing development. These, and other such directly-encoded effects, led directly to the "calcification" of the autonomous driving components discussed in the next chapter, as seemingly small updates to include "just one more" status flag, or to tweak "just one more" default calculation became increasingly difficult to analyze, implement, and verify.

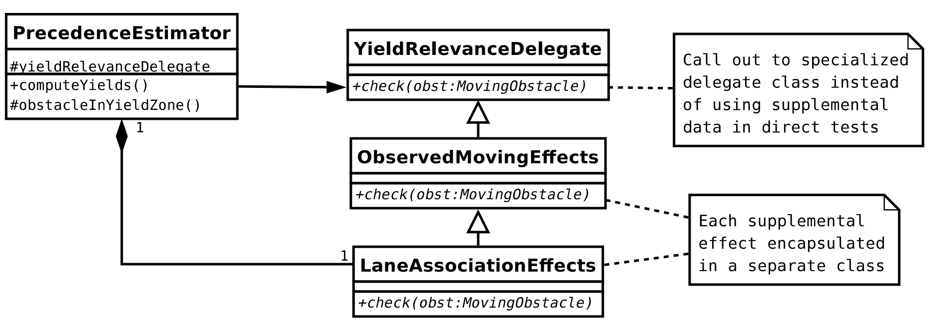


Figure 4.3: Object-Oriented delegation of the yield relevance test to encapsulate supplemental effects according to the Decorator [21] pattern. Problems typically associated with long inheritance chains are mitigated using Mixin Layers[51], as shown in Listing 4.2.

To address these issues, and to provide a degree of flexibility on par with the Aspect-Oriented design presented in Section 4.3, the OO approach evaluated in this thesis delegates the decision of "yield relevance" to a small Decorator[21] class that may be specialized to introduce various supplemental effects. The UML diagram in Figure 4.3 shows a simplified view of how the individual effects of the "isMoving", "isObservedMoving" and "laneAssociations" supplemental data may be isolated according to this pattern.

One common criticism of the pure Decorator pattern is that long inheritance chains make it difficult to extract or rearrange the precedence of individual effects. To mitigate this issue, the classic Decorator pattern has been enhanced by the use of Mixin Layers [51], as shown in the pseudo-code in Listing 4.2. To a certain extent, this may also have been addressed through the use of an alternate OO design pattern, such as Chain of Command[21], and while this and other patterns are perfectly valid approaches to the problem, the fundamental mechanism of delegation to external classes remains the same, and informal experiments indicate that there would be no significant difference in the metric results presented in Chapters 6 and 7.

```
// default case for YieldRelevance is "true"
class YieldRelevanceDelegate {
  virtual bool check(MovingObstacle obst) {
    return true;
  }
};
// each supplemental datum gets its own class
// inheritance chaining is done using template trickery of Mixins
template <class PARENT> class ObservedMovingEffects: public PARENT {
   virtual bool check(MovingObstacle obst) {
     return obst.isObservedMoving && PARENT::check(obst);
  }
};
// and for Lane Associations
template <class PARENT> class LaneAssociationEffects: public PARENT {
   virtual bool check(MovingObstacle obst) {
     return !obst.laneAssociations.empty() && PARENT::check(obst);
  }
};
// composition of yieldRelevanceDelegate from Mixin components:
class PrecedenceEstimator {
  // ...
  LaneAssociationEffects < ObservedMovingEffects <
    YieldRelevanceDelegate > > yieldRelevanceDelegate_;
};
// call out to delegate class instead of using supplemental data
void PrecedenceEstimator::computeYields() {
  // ...
  if(obstacleInYieldZone(obst) &&
     yieldRelevanceDelegate_.check(obst)) {
    // main body of yield calculation...
  }
}
```

Listing 4.2: Pseudo-code showing object-oriented delegation of the “yield relevance” test.

In this alternate design, the core Precedence Estimator algorithm remains embedded in the `PrecedenceEstimator` class. This class no longer contains the direct dependencies on the *supplemental* “observed-moving” or “lane associations” data from Listing 4.1, fulfilling Requirement **AR.2**. Supplemental effects are bound to the core algorithm by extending the `YieldRelevanceDelegate` class instead of directly modifying the `computeYields()` method, which satisfies Requirement **AR.3**, both in terms of eliminating "invasive modification" and by eliminating the need to understand the "inner workings" of the core `PrecedenceEstimator` implementation. Lastly, the effects of each supplemental datum are encapsulated in individual classes, and those effects can be removed by simply excluding the corresponding class from the inheritance chain for `YieldRelevanceDelegate`, satisfying Requirement **AR.4**.

4.3 Aspect-Oriented Technique: XPI

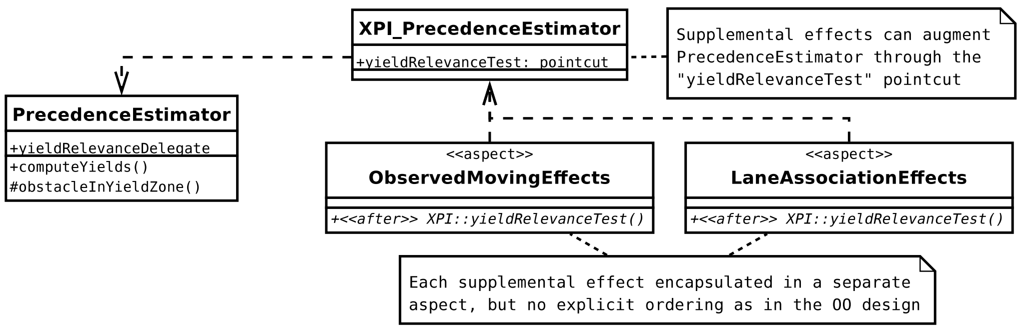


Figure 4.4: Aspect-Oriented exposure of adaptability through a Crosscutting Programming Interface (XPI), and binding supplemental effects as "after" advice through the XPI

Following discussion in the previous chapter, the second design technique uses a Crosscutting Programming Interface[24] (XPI) to expose an algorithm's "likely points of variability", allowing individual supplemental effects to be bound to the core algorithm using AO "advice" directives, as illustrated in Figure 4.4 and Listing 4.3.

The most notable differences in the AO design for encapsulating supplemental effects are that the core algorithm does not need to call out to an explicit delegation interface, and the individual supplemental effects only depend on the XPI, instead of each other. These lead to a much more concise description of individual supplemental effects, imposing less "overhead" for introducing or removing those effects as compared to the OO design, which are drawn out in the analytical results presented in Chapters 6 and 7.

In terms of the requirements laid out in Chapter 2, the core algorithm remains within the `PrecedenceEstimator` class, which is free of direct dependencies on supplemental data as with the OO design above, satisfying Requirement **AR.2**. Requirement **AR.3** is addressed by `XPI_PrecedenceEstimator`, which exposes the "points of variability" in the core algorithm as named "pointcuts", such as `yieldRelevanceTest`. Lastly, the effects of individual supplemental data are implemented in separate *aspects* that can be introduced or removed even more easily than their OO counterparts, fulfilling Requirement **AR.4**.

One critical drawback to this approach is that, at the time of this writing, AO techniques in general are relatively new, so developers are less likely to be familiar with the corresponding tools and methodologies. Also, relative to the particular experiment outlined in the following section, the requisite AspectC++[20] weaver is less mature and less well-supported than its more Java-centric counterparts[10]. Still, both of these issues are

```
void PrecedenceEstimator::computeYields() {
  // set up as before, but no extra calls in relevance test
  if(obstacleInYieldZone(obst)) {
    //  main body of yield calculation...
  }
}
// XPI exposes adaptability in core algorithm
aspect XPI_PrecedenceEstimator {
  // expose the in-yield-zone method for AO advice introduction
  pointcut yieldRelevanceTest(obst) =
    execution ("obstacleInYieldZone") && args(MovingObstacle obst);
}
// each supplemental datum gets an aspect
aspect ObservedMovingEffects {
  // augment yield relevance test to require obst->isObservedMoving
  advice XPI_PrecedenceEstimator::yieldRelevanceTest(obst) : after()
  {
    // tjp (The Join Point) allows manipulation of the return value
    *(tjp->result()) &= obst.isObservedMoving;
  }
}
// but there is no explicit composition as in OO design
aspect LaneAssociationEffects {
  // augment yield relevance test
  advice XPI_PrecedenceEstimator::yieldRelevanceTest(obst) : after()
  {
    *(tjp->result()) &= !obst.laneAssociations.empty();
  }
}
```

Listing 4.3: Application of supplemental yield-relevance effects through an XPI. See Appendix B for an extended presentation of AO syntax and concepts.

likely to erode over time as AO tools and techniques become more mainstream, and, in the context of this thesis, AO techniques are certainly worth investigating as an alternative to the now-classic OO techniques outlined above.

4.4 Experimental Outline

The two detailed design techniques described above have been experimentally applied to existing software for autonomous driving behaviors in order to evaluate the proposed primary vs. supplemental methodology. Chapter 5 reviews the overall software system, then presents the detailed refactoring process and results for each of three distinct components. This process yielded a total of nine artifacts for analysis: the original implementation, plus an AO and OO design for each such component.

One of the difficulties in evaluating the relative merits of different software designs is that software engineering is focused on achieving various *qualities* in a given software system, which are intrinsically difficult to *quantify* for concrete comparison. Thus, such comparisons are often highly anecdotal, conveying the personal experiences of a given author and the various difficulties he may or may not have faced in trying to develop or maintain the software in question. While it is possible to convey (and glean) tremendous insights from this kind of discussion, it is still important to provide a concrete foundation for comparison in terms of measurable properties of the software. Even though such metrics do not directly convey pure qualities, such as the *adaptability*, of a software system, an appropriate selection of them, combined with careful discussion of their relative values for each candidate design, can provide compelling evidence of the merits of each technique.

As the detailed design techniques discussed above include Aspect-Oriented methodology, this work focuses on evaluation techniques that have been shown to effectively display the relative merits of AO and OO designs for the same system. Of these, two[34, 22] stand out in particular as providing compelling evidence for the benefits and tradeoffs to be had between AO and OO techniques. The principal metrics from these two studies have been applied to the original and refactored artifacts described above, including both source-level "Concern Diffusion" metrics[35], presented in Chapter 6, and design-level "Net Option Value" analysis[5, 55], presented in Chapter 7.

To further explore the methodology proposed in this thesis, candidate extensions to novel input data, drawn from "vehicle-to-vehicle" communications standards[15], are discussed in Chapter 8. Several candidate adaptations are closely analyzed in terms of how well the refactored designs accommodate the necessary changes, yielding additional insights into the identification of "likely points of variability" in advance that will accommodate many future supplemental effects.

Thereafter, the lessons garnered throughout the refactoring and analysis of autonomous driving software are extended into the domain of planetary (Mars) exploration in Chapter 9. In particular, candidate supplemental data are identified in the CLARAty[38] implementation of Morphin[59], and alternate designs are proposed in accordance with the techniques described in Sections 4.2 and 4.3. The ready identification of candidate supplemental data, and the discussion of the corresponding designs together support the applicability of the proposed methodology beyond the scope of autonomous automobiles. Insights gleaned from both case studies are then aggregated into guidelines for how to apply the proposed *primary* vs. *supplemental* methodology to novel systems and other problem domains in Chapter 10.

Chapter 5

Application to Autonomous Driving Behaviors

This chapter describes the refactoring of existing software from the Behavioral Executive subsystem of Boss, Tartan Racing's winning entry to the 2007 DARPA Urban Challenge. Section 5.1 begins with a brief description of this subsystem, concentrating on the moving obstacle representation and its three dependant components, which together form the focus of this experimental analysis. Supplemental data in the moving obstacle representation are identified, followed by a detailed analysis in Sections 5.2, 5.3 and 5.4 of the algorithm embodied in each component, including discussion of the overall algorithm, identification of supplemental effects therein, and presentation of each of the AO- and OO-refactored designs. The overall refactoring process, including discussion of issues common to all three components, is summarized in Section 5.5 before proceeding to the analytical results presented Chapters 6 and 7.

5.1 Autonomous Driving Behaviors in the Urban Challenge

The Urban Challenge[16] was an autonomous vehicle competition sponsored by the US Defense Advanced Research Projects Agency (DARPA) in 2007. Contestant robots were required to autonomously execute a series of navigation missions in a simplified urban environment consisting of roads, intersections, and parking lots while obeying road rules and interacting safely and correctly with other traffic. In contrast to the two preceding Grand Challenges[25, 26], which focused on rough-terrain navigation, the urban competition required the development of a system capable of complex traffic behaviors such as waiting for precedence at an intersection or passing a slow-moving vehicle on a multi-lane road.

These behaviors were managed by a software subsystem called the *Behavioral Executive* in Boss, Tartan Racing's winning entry in the Urban Challenge. Within the Tartan Racing software system[58], the Behavioral Executive was responsible for generating a sequence of incremental goals for execution by the underlying motion planner, and for the modulation of those goals to conform to the Urban Challenge rules for vehicle interaction. Typical goals included driving to the end of the current lane or maneuvering to a particular parking spot, and their issuance was predicated on conditions such as precedence at an intersection or the detection of certain anomalous situations. In the case of driving along a road, periodic

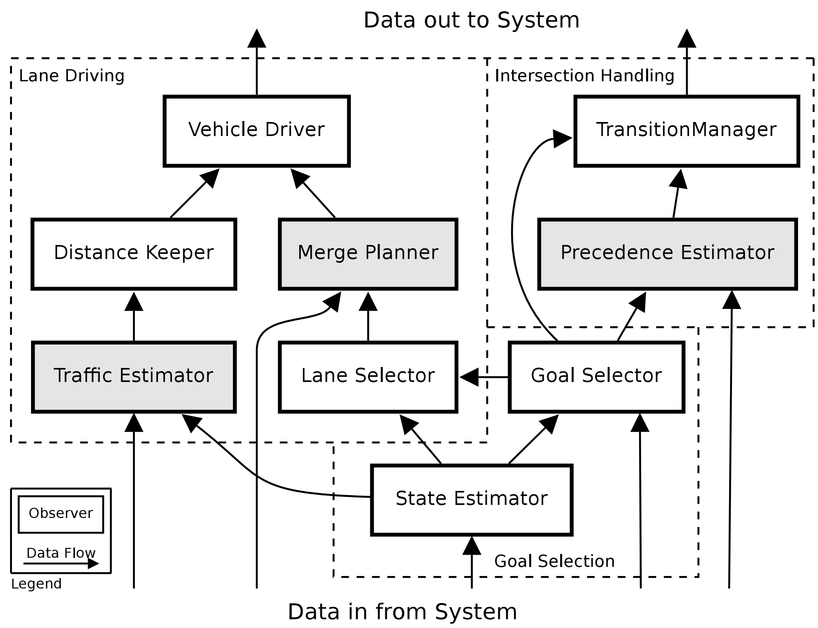


Figure 5.1: Abstract data flow view of the Behavioral Executive, showing dominant elements and data paths, grouped by functional context. Shaded Observers became particularly dependent on the system's representation of moving obstacles.

lane tracking and speed government commands were published to enact behaviors such as safety gap maintenance, passing maneuvers and queueing in stop-and-go traffic.

As presented in [3, 4], the Behavioral Executive made significant use of standard object-oriented design patterns, such as the Adapter, Strategy, Factory, and Observer[21] patterns in order to remain adaptable to the evolving needs of the competition development. In essence, the Behavioral Executive was decomposed into a collection of independent classes, called *Observers* that communicate through modifications to and subsequent change-notifications from a collection of persistent, intermediate data elements called *Subjects*. Each Observer fulfilled a specific responsibility within the subsystem, such as goal selection or distance keeping, and new functionality could be easily introduced by adding, replacing or extending individual Observers. The principal functionality of the Behavioral Executive was implemented in nine Observers, which can be grouped into three functional contexts as shown in Fig. 5.1.

While there were, in fact, many more functional elements and more convoluted data paths than illustrated, they generally belong to auxiliary functionality, such as diagnostic state reporting, and are omitted for clarity. Those that remain represent the three most abstract responsibilities of the Behavioral Executive:

- The selection of incremental goals along the path from the robot's current location to the next checkpoint (Goal Selection);

- The conditional transmission of those incremental goals according to whether it is the robot's turn at an intersection (Intersection Handling);

- The continuous management of various behaviors such as distance keeping and passing maneuvers on multi-lane roads (Lane Driving).

Each of the Observers that comprise the Behavioral Executive encapsulates an algorithm that reasons about the data available from the robot's perception subsystem in order to govern the actions of the robot. While the results of the competition certainly underscore the effectiveness of the Behavioral Executive as-implemented, the design nevertheless showed several weaknesses over the course of the development. In particular, the decomposition into individual Observers, while allowing the wholesale substitution of alternate functionality, did not in itself treat the problem of adaptability to changes in input data. This became more problematic as the development proceeded, as many algorithms came to depend very heavily on the specific content and semantics of the data available from the perception subsystem, especially the representation of other traffic, the so-called `MovingObstacle` class, shown in Figure 5.2.

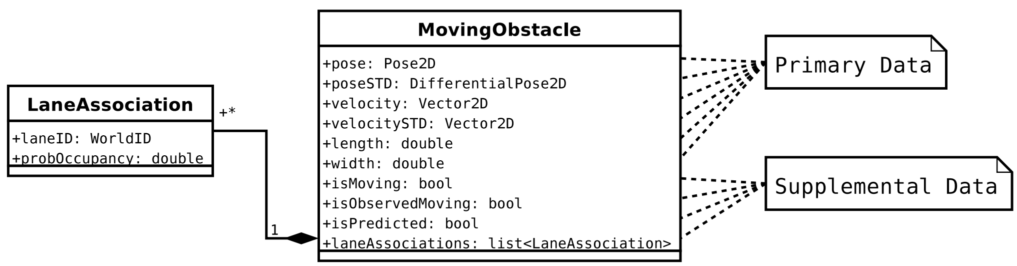


Figure 5.2: UML representation of the `MovingObstacle` class, which preserves semantic content but is otherwise simplified from the as-implemented version.

It is important to recognize that this `MovingObstacle` representation is somewhat simplified from the actual class signature used for the Urban Challenge. For instance, the "true" version guarded access to the member data listed in Figure 5.2 behind traditional accessor and mutator methods, such as `GetObservedMovingFlag()` and `SetObservedMovingFlag()`. Rather than inflate the representation in this manner, these and similar data are represented by public members such as `isObservedMoving` for clarity and brevity. The "true" version also included tertiary functionality, such as prediction of future poses, along with support for serialization[1], time-stamping, visualization, and debugging, all of which were unused by the Behavioral Executive. These are also excluded from the simplified `MovingObstacle` representation, leaving only the raw data that is relevant to this discussion.

The complete set of detectable traffic was provided as a list of `MovingObstacle` instances that was used by exactly three Observers, the Traffic Estimator, Precedence Estimator, and Merge Planner, which are highlighted as gray boxes in Figure 5.1. In the original designs for these components, little or no effort was put into isolating the effects of individual data, and the team relied instead on intimate familiarity with the associated source code to implement changes on an as-needed basis. Subsequently, the accumulating dependence on the presence and precise meaning of several volatile properties of the `MovingObstacle` representation made these components particularly brittle relative to ongoing changes in the perception subsystem and/or the underlying platform. Of the member data listed in

[1]Serialization is a software engineering term for "conversion to a network-transportable format".

Figure 5.2, only the first three: pose, velocity, and size, were semantically stable throughout the development, and, according to guidelines presented in Chapter 4, these will be treated as *primary* data. The other four will be treated as *supplemental* data, as they were incrementally introduced as the perception system became more capable, and their semantics were frequently revised as underlying sensors and modeling techniques were substituted or enhanced:

- **Is Moving**, which indicates that the obstacle is currently believed to be in motion, in an abstract sense beyond simply having an instantaneously nonzero velocity;
- **Is Observed Moving**, which indicates that the obstacle's historical motion has been consistent with the perception subsystem's model of an actual automobile[2];
- **Is Predicted**, which indicates that the obstacle's position, velocity and even existence are based entirely on extrapolation from historical observations, and are not supported by any immediate observations;
- **Lane Associations**, which provides a probabilistic list of road lanes that the obstacle could plausibly be tracking, given its observed motion and behavior over time.

Note that none of these directly implies "this is *really* a car", but they instead provide extra information that should be interpreted according to the current context to determine whether the obstacle is worth considering, and how to subsequently treat it, further reinforcing their *supplemental* treatment. The extent of their effects can be roughly estimated by counting references in the source code, either directly to the corresponding fields of the `MovingObstacle` class, or else indirectly through some intermediate or cached result, as summarized in Table 5.1.

Obstacle Property	Direct References	Indirect References
Is Moving	2	35[3]
Is Observed Moving	8	37[3]
Is Predicted	1	1
Lane Associations	5	0
Total	16	38

Table 5.1: Supplemental moving obstacle data references in the original implementation of the Tartan Racing Behavioral Executive.

The 16 direct and 38 indirect references to these data are scattered across three behavioral components mentioned above:

- **Traffic Estimator**, which is responsible for identifying, of all candidate vehicles in perception range, the one most likely to be "in front" of the host vehicle in the current lane;

[2]Note that an "Observed Moving" obstacle can be temporarily stopped, and so does not require that the obstacle be instantaneously "Moving" as well.

[3]There are 35 references to an intermediate result that depends on both "Is Moving" and "Is Observed Moving" which would otherwise be double-counted in the total

- **Precedence Estimator**, which is responsible for determining the precedence ordering at intersections, waiting for the intersection to be free of vehicles, and yielding to any cross traffic before proceeding through the intersection;
- **Merge Planner**, which is active when a lane-change is commanded and is responsible for identifying, synchronizing with, and merging into openings between vehicles in the target lane.

The complexity of these components, and the nature of the supplemental effects therein, make them good candidates for testing the effectiveness of the proposed techniques. As such, these components are the focus of the experimental work at the core of this thesis, and they have been refactored according to the OO and AO design techniques described in Sections 4.2 and 4.3. The remainder of this chapter reviews the algorithm embodied in each component, including a detailed discussion of the supplemental effects found therein, and presents the refactored designs that separate those effects from their associated core algorithms. Raw counts of files, classes, and lines of code for each refactored artifact are presented and discussed as precursors to the more advanced metrics in Chapters 6 and 7.

5.2 Traffic Estimator

The Traffic Estimator is responsible for identifying, among all candidate obstacles, the leading vehicle (if any) in front of Boss, and providing a conservative estimate of both the distance to that vehicle and how fast it is travelling along the road. This component made use of several data, as illustrated in Figure 5.3, including the list of candidate moving obstacles, the estimated position and speed of the host platform and the known geometry and connectivity of the road network. The latter two are ignored for the purposes of this case study, which focuses on the contents of the `MovingObstacle` representation, especially on the effects of the supplemental data listed above.

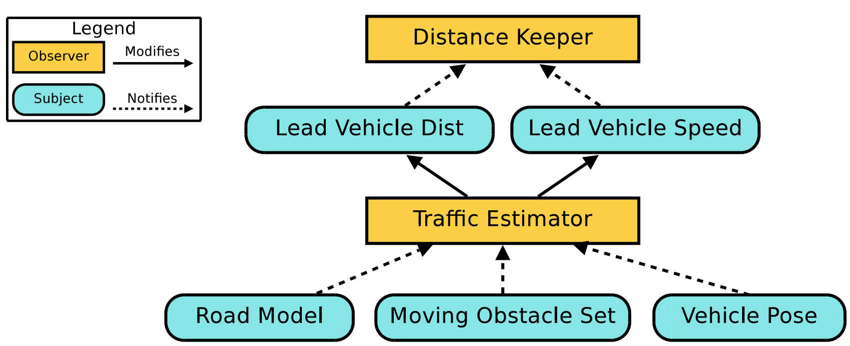


Figure 5.3: Traffic Estimator collaboration diagram, showing critical input and output Subjects according to the Observer[21] pattern.

All four supplemental moving obstacle data had some effect on the Traffic Estimator's algorithms, and the relative simplicity of their influence provided a good starting point for working out the implementation details of each design. In the language of software requirements, these supplemental data had the following effects:

TE.S.1 Regardless of the instantaneous velocity of the lead vehicle, it shall be treated as though it is stopped at its current location unless its `isMoving` property is set to *true*.

TE.S.2 Regardless of the instantaneous velocity of the lead vehicle, it shall be treated as though it is stopped at its current location unless its `isObservedMoving` property is set to *true*.

TE.S.3 Regardless of the instantaneous velocity of the lead vehicle, it shall only be considered as having negative (oncoming) speed if the associated `MovingObstacle` instance backed by current sensor observations, as indicated by its `isPredicted` property being set to true.

TE.S.4 Regardless of the instantaneous velocity of the lead vehicle, it shall only be considered as having negative (oncoming) speed if it is strongly associated with a single lane, as enumerated by its `laneAssociations` property. Whether this effect is active shall be load-time configurable.

TE.S.5 Instead of the default geometric checks for whether an obstacle is in Boss's current lane of travel, the Traffic Estimator shall check its `laneAssociations` list against a minimum threshold to determine occupancy of a given lane. This threshold shall be load-time configurable.

The first four effects are all interrelated in that they alter the estimation of the lead vehicle's speed to be more conservative in dubious or critical situations. In a sense, they specify the conditions under which the velocity information should be "trusted", and what constitutes a "safe" alternative speed when the velocity information cannot be "trusted". In this case, "mistrust" leads unilaterally to an estimated speed of zero, which in turn leads to more conservative behavior in the consumers of the "Lead Vehicle Speed" output, such as the Distance Keeper.

The first two effects, **TE.S.1** and **TE.S.2**,bind "trust" to the ideas of directly-measured velocity (i.e., via RADAR) and plausible historical motion, respectively. If either of these conditions is not met, then it is deemed "safer" to treat the obstacle in question as though it is stopped at its present location, regardless of its reported velocity. The latter two effects, **TE.S.3** and **TE.S.4**, require the obstacle to be backed by recent sensor readings and to be travelling in a single, unambiguous lane before "trusting" a negative lane speed. In this case, negative speed implies "oncoming" traffic to the Distance Keeper, which could, in turn, trigger a risky off-road "evasive maneuver" to avoid an impending head-on collision. In this case, the `isPredicted` and `laneAssociations` data are applied as additional safeguards to be *as sure as possible* that the evasive maneuver is warranted.

One common approach to dealing with variations in such platform-specific contributors to the idea of "trust" in sensor data is to hide them all behind a more abstract notion of "confidence" or "certainty", such as shown in Figure 5.4. In this hypothetical case, the four *supplemental* moving obstacle data listed above are combined into a single scalar `sensorConfidence` value[4] that is meant to be shared by all consuming algorithms.

[4]Such abstract "confidence" values are often unit scalars (i.e., limited to the span $[0, 1]$), and are often expressed as "accuracies" or "variances" that are simply "fudged" to express data such as `isObservedMoving`.

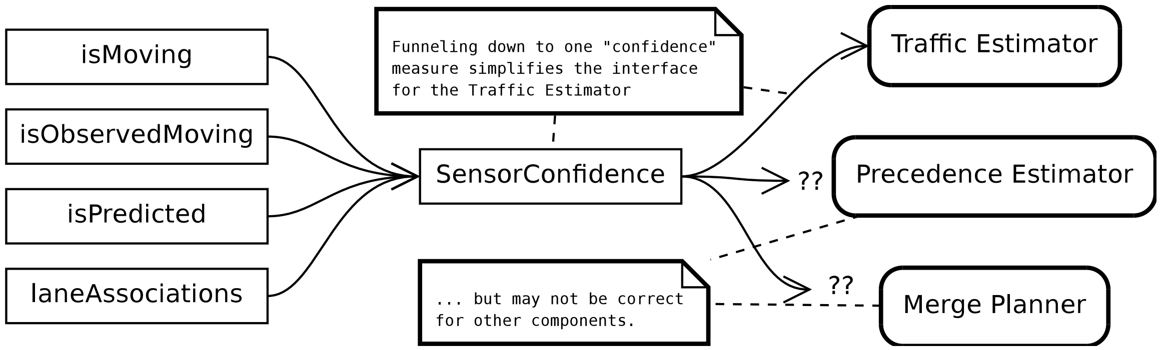


Figure 5.4: The trouble with abstract "confidence" values: a single abstract value may not be sufficiently expressive to allow the full range of desired effects in consuming algorithms.

The principal issue with this single `sensorConfidence` value is that, while it may easily be engineered to express the requirements for the Traffic Estimator listed above, it is difficult to guarantee that the particular algorithm for converting `isObservedMoving`, etc. into a scalar "confidence" value will simultaneously satisfy related, but not identical requirements in other components. For example, requirement **PE.S.3** for the Precedence Estimator directly couples the `isObservedMoving` property to a particularly context-specific notion of a "busy" intersection, which would likely be expressed as a threshold on the `sensorConfidence` value. In order to not "accidentally" trip such thresholds as a function of other, irrelevant *supplemental* data, knowledge of each downstream threshold would have to be embedded in the algorithm that generates the `sensorConfidence` value. From a software design perspective, this would have two particularly undesirable effects:

1. The producing algorithm would be strongly coupled to its various consumers, which limits the reusability each individual component;

2. Through the shared "knowledge" of such thresholds, underlying properties such as `isObservedMoving`, etc. are still *implicitly* expressed in the abstract `sensorConfidence` value, which erodes the original benefits of "hiding" such values behind a more abstract representation.

To further compound the issue, supplemental data can have other effects beyond the scope of an abstract "confidence" measurement.. For instance, **TE.S.5** describes the replacement of a small segment of geometric reasoning in the core Traffic Estimator algorithm with a direct dependency on the `laneAssociations` supplemental datum. In this case, there is no analogue to a "confidence" value that will enable this functionality, which further underscores one of the fundamental premises of this thesis: that no single input data representation can sufficiently cover the broad applicability of context- and platform-specific data.

The critical issue at hand is that the determination of "trust", "confidence", etc., in the information provided by a robot's perception system is a function of both the data provided by the robot's constituent sensors and the specific algorithmic contexts in which those data are to be interpreted. The designer's role in this matter is to choose whether to place the "interpretation" of platform-specific sensor data:

1. **Close to the sensors**, such as by using the abstract `sensorConfidence` value discussed above,

2. **Close to the consuming algorithms**, such as by forwarding all potentially-relevant sensor data to all consuming algorithms, or

3. **Somewhere in-between**, such as by keeping some context-free interpretation "close to the sensors", and leaving any remaining context-specific interpretation to be done "close to the algorithms".

The "middle ground" described by #3 above implies a spectrum of tradeoffs that a designer must consider when determining where to place the various "interpretive" stages of a robotic software system. As illustrated in Figure 5.5, keeping all sensor data interpretation "close to the sensors" maximizes the stability of the interface to other components in the system at the cost of limited expressivity and the potential for "hidden" couplings to consuming algorithms. At the other end of the spectrum, keeping all interpretation "close to the consuming algorithms" maximizes flexibility in specifying the detailed and unique effects of each sensor datum on each algorithm, but it also maximizes the volatility of the interface between the perception system and advanced reasoning algorithms, the latter of which will be more sensitive to changes in the underlying platform.

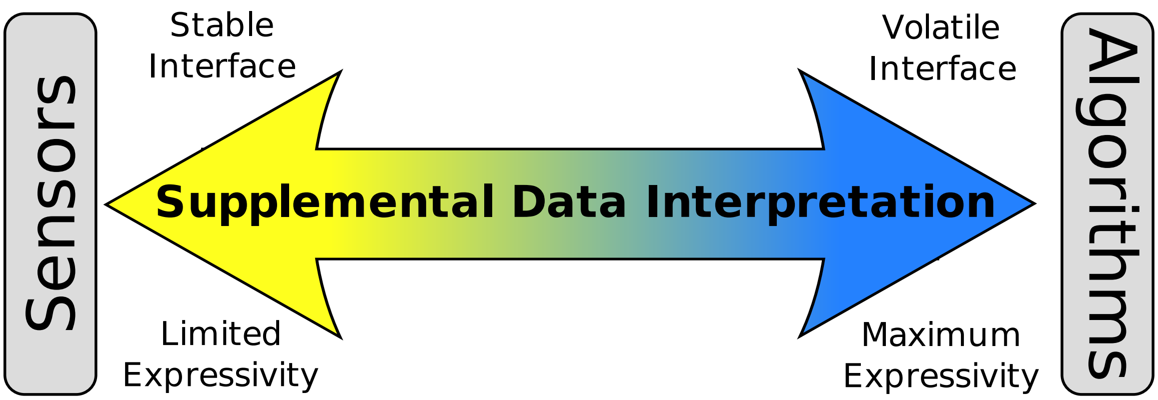


Figure 5.5: The critical tradeoffs a designer must make when choosing to keep interpretation "close to the sensors" vs. "close to the consuming algorithms".

This thesis focuses on exploring this "middle ground", which is where the results of a system's context-free interpretation of sensor data (`isMoving`,`isObservedMoving`, etc.) are folded into context-specific reasoning process (autonomous driving behaviors). In so doing, the overarching goals of this thesis are to contribute nomenclature, candidate designs, and analytical results thereof in order to guide future designers in reasoning about similar tradeoffs in their own systems.

Redesign Experiments

The original version of the Traffic Estimator was encapsulated in a single class, and it encoded the five effects listed above in-line with the main algorithm, with no specific treatment given to the types of adaptability laid out in Chapter 2. The first four effects are all one- or two-line introductions within nested logic statements surrounding the estimation of the lead vehicle speed, which are summarized in Listing 5.1. Note that this is simplified pseudo-code: the original implementation was roughly twice as large, and the logic therein was somewhat more convoluted, as the "core" algorithm for speed estimation included additional geometric checks as to whether or not "small" or "negative" speeds were permissible. The extra logic wrapped around this otherwise simple method is an example of the "calcification" mentioned earlier, as the adaptation of this algorithm requires the developer to absorb and understand the entire logic structure before inserting new effects or updating or removing existing effects. The alternate designs, presented below, instead isolate the individual effects in separate classes, allowing them to be added, removed or updated separately.

```
double TrafficEstimator::estimateObstacleSpeed(MovingObstacle &mo)
{
  double laneSpeed_mps; // computed result for this method

  // verify that we "trust" the velocity vector
  if( mo.isMoving &&           // *** TE.S.1
      mo.isObservedMoving )    // *** TE.S.2
  {
    // do the "normal" lane speed calculation
    // note: this is the "core" algorithm for speed estimation
    laneSpeed_mps = projectVelocityVectorOntoLaneHeading();
    if(laneSpeed_mps < 0.0)
    {
      // verify that negative/oncoming velocity is allowed
      if(mo.isPredicted ||     // *** TE.S.3
          (oncomingRequiresStrictLaneAssociation_ && // *** TE.S.4
           mo.laneAssociations.size() != 1)          // *** TE.S.4
        )
      {
        // oncoming not allowed: force to zero to be safe
        laneSpeed_mps = 0.0;
      }
    }
  } else {
    // not trusted: force to zero to be safe
    laneSpeed_mps = 0.0;
  }

  return laneSpeed_mps;
}
```

Listing 5.1: Pseudo-code for supplemental effects in Traffic Estimator speed estimation.

The last requirement, **TE.S.5**, supplanted geometry-based determination of lane occupancy with direct usage of the `laneAssociations` datum. From a certain perspective, this datum might be expected from any reasonably capable autonomous driving perception system, making its *supplemental* treatment somewhat debatable. Indeed, if it were specified at the beginning of the development process and made available right away, it may have been treated as a *primary* datum instead. Neither of these were the case, however, as the `laneAssociations` listing was not part of the original conception of the `MovingObstacle` representation, nor was it introduced until very late in the development process. These forced the development and use of more generic, geometry-only functionality, which was eventually supplanted when the `laneAssociations` listing was introduced as an *additional* datum. In fact, the underlying functionality had been completely removed from the Traffic Estimator, and it was necessary to delve into the history of the software repository to recover the original geometry-based algorithm. Without access to this version history, porting the Traffic Estimator to a system that lacked the `laneAssociations` datum would be highly problematic, as the core algorithm would be suddenly incomplete.

To avoid this particular difficulty, the OO and AO designs instead leave the baseline geometric algorithm in place, encoding the *suppression* of that underlying functionality as part of the declaration of the `laneAssociations` supplemental effects.

Object-Oriented Design

The OO redesign, shown in Figure 5.6 of the Traffic Estimator followed the pattern described in the previous chapter without deviation, with the single exception of the creation of a `GenericDelegate` class to provide some simple support for configuration and initialization. All delegate interface classes in all subsequent OO designs inherit from this class, so this dependency is only shown in this first diagram. For additional clarity and brevity, explicit dependencies on `MovingObstacle` data are omitted in this and all subsequent UML diagrams, relying instead on appropriate naming of supplemental delegates to infer data dependencies. For example, the class `OOTE_ILT_LaneAssociationEffects` class applies the `laneAssociations` *supplemental* datum through the `InLaneTestDelegate` interface in order to fulfill **TE.S.5**.

This brings another minor issue into the foreground: the application of such small effects in such specific contexts poses curious challenges to the appropriate naming of supplemental effect classes. More specifically, a well-named supplemental delegate should describe the class that embodies the core algorithm, the application context of the supplemental effect, and the supplemental data that contribute to the effect. Given nontrivial names for each of these, supplemental delegates would have names that easily exceed 50 characters, which is untenable. Instead, abbreviations are used for the first two, such as in the class from the previous example, `OOTE_ILT_LaneAssociationEffects`, which stands for:

- **OOTE** Object-Oriented [OO]TrafficEstimator;
- **ILT** InLaneTest[Delegate];
- **LaneAssociationEffects** Application of the `laneAssociations` supplemental datum.

This naming scheme is reasonably concise, unambiguous, and, given appropriate context in each diagram, easy to resolve by following inheritance or dependency paths as necessary. Thus, this scheme has been adopted for all class names in all subsequent diagrams, both for the OO and AO designs.

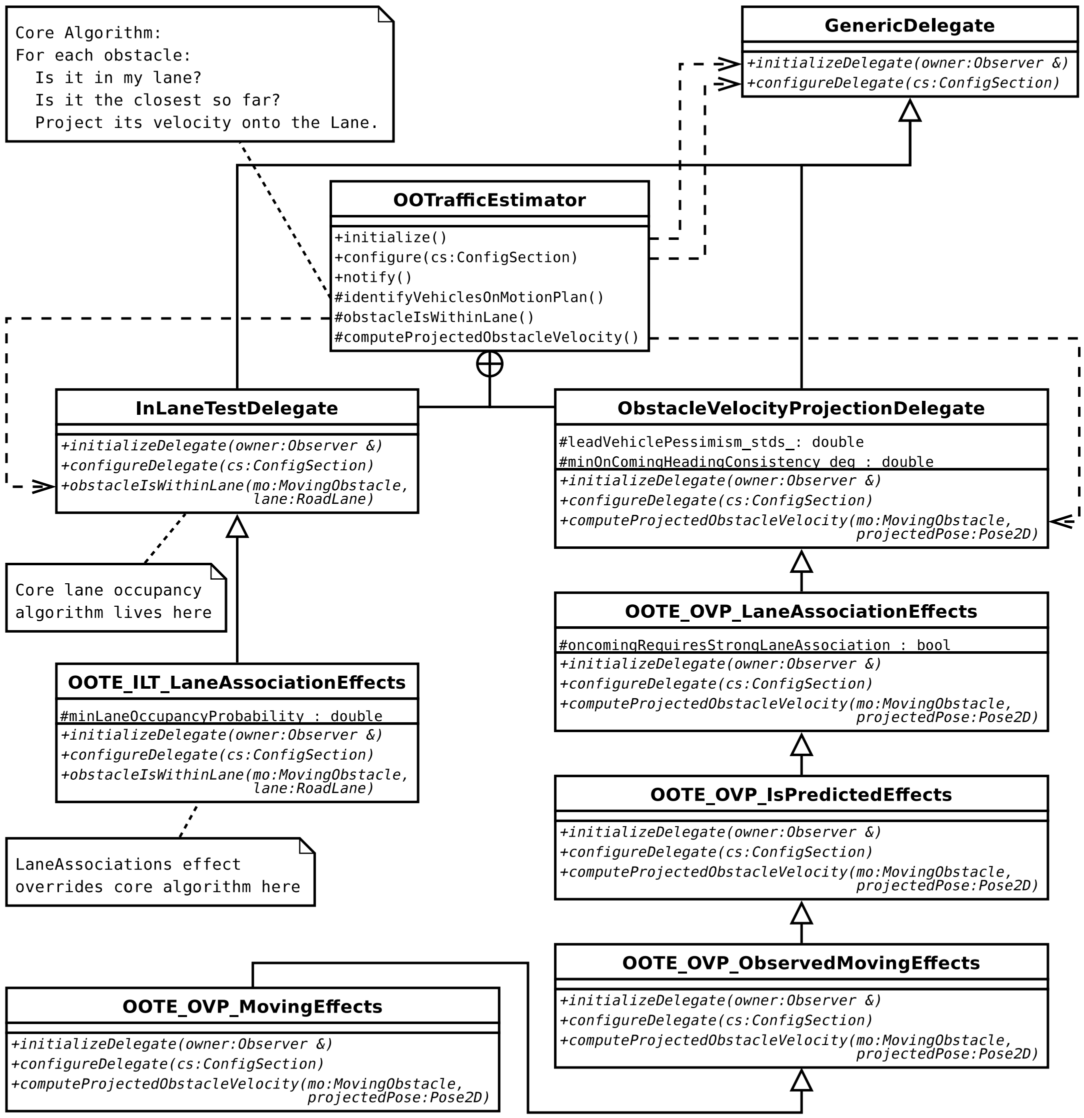


Figure 5.6: Object-Oriented redesign of the Traffic Estimator, isolating supplemental effects in separate delegate classes. The ⊕ symbol represents nested typing, see Appendix E.5.

Following this scheme, it is easy to see that the other four requirements, **TE.S.1** through **TE.S.4**, are each encapsulated in a separate class, yielding a chain of specialization against the `ObstacleVelocityProjectionDelegate` interface. Each class in this chain overrides the *computeProjectedObstacleVelocity()* method as is necessary for each effect. The order of specialization controls the order of priority, with the bottom-most class, `OOTE_OVP_MovingEffects` having the highest priority. The depth of this inheritance tree, combined with the fact that each delegate must explicitly call up to its parent, leads to a collection of dependencies and an explicit ordering of operations that are one of the more significant weaknesses of the OO design. Even though a technique similar to the idea of "mixins" from [51] has been used to limit the number of explicit dependencies, there must always be at least one component that "knows" about all of the supplemental delegates in order to compose them into the runtime instance used by the OOTrafficEstimator.

For effects such as **TE.S.1** through **TE.S.4**, which all have the same "conservative" effect of overriding the speed estimate to zero, no strict ordering is necessary, which leaves the OO design with "unnecessary" interdependencies. One of the principal differences between this design and the AO design presented in the next section is that the AO design will allow these effects to remain "oblivious" to each other, yielding a somewhat less constrained and more concise implementation of the same functionality.

Aspect-Oriented Design

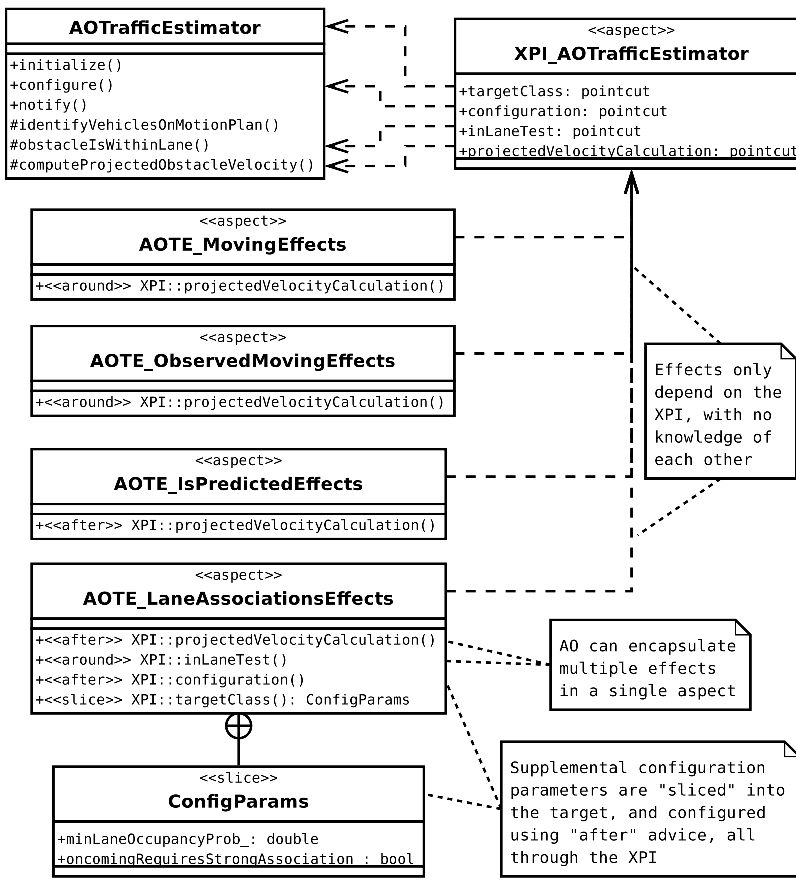


Figure 5.7: Aspect-Oriented redesign of the Traffic Estimator, isolating supplemental effects in separate aspects through a crosscutting programming interface. The "slice" stereotype on the `ConfigParams` class represents the inter-type declaration mechanism in AspectC++, see Appendix E.7.

As with the OO version, the AO redesign of the Traffic Estimator, shown in Figure 5.7 followed the pattern established in the previous chapter without deviation. A crosscutting programming interface, `XPI_AOTrafficEstimator` exposes named pointcuts in the `AOTrafficEstimator` implementation to allow variation of the core algorithm. Many of these pointcuts have direct analogues in delegates in the OO design, such as the *configuration* pointcut, which is analogous to `GenericDelegate::configureDelegate()`, and the *inLaneTest* pointcut, which is analogous to the role of the `InLaneTestDelegate` interface from Figure 5.6. These similarities are largely due to the fact that these two designs are trying to solve the same problem.

The AO design differs, however, in that these pointcuts are accessible to any aspect in the system, without the need to create individual classes for each individual effect. This can be seen in the `AOTE_LaneAssociationsEffects` aspect, which applies advice to both:

- **XPI::projectedVelocityCalculation**, in fulfillment of **TE.S.4**, and
- **XPI::inLaneTest**, in fulfillment of **TE.S.5**.

This allows the effects of any one datum to be grouped within a single source-level construct, making it very easy to review the collected effects thereof, should the meaning of the datum be *altered*. When combined with the advantage of "obliviousness" discussed above, this also reduces the extraction of the effects of an *absent* datum to be a simple act of modular exclusion. In this case, simply deleting the file that contains `AOTE_LaneAssociationsEffects`, then re-weaving the system, will yield an `AOTrafficEstimator` that is independent of the `laneAssociations` supplemental datum.

This contrasts with the OO design, wherein the explicit composition of supplemental effects, discussed above, induces some overhead beyond simply deleting the file to perform a similar extraction. This additional overhead shows up as additional "scattering" of the supplemental effects in the Concern Diffusion metrics presented in the next chapter.

The AO design is not without its faults, however, which are highlighted by the many method-level dependencies of the XPI on the core implementation. Beyond the fragility of these dependencies, as discussed in Section 3.2, it was necessary to refactor the core algorithm in several specific ways in order to allow AO *advice* introductions. Specifically, each point in the core algorithm where supplemental effects might be applied had to be factored out as a separate method, such as the *obstacleIsWithinLane()* method in Figure 5.7, in order to allow AO interception of the corresponding functionality. While such method-level refactoring might provide some benefit to the understandability of the core algorithm, it can also lead to the introduction of many trivial functions, as was the case for the Precedence Estimator, discussed in Section 5.3.

From a certain perspective, this may make the OO design somewhat more attractive from a designer's perspective, as it does not require any specific structure in the core algorithm and simply "calls out" to the specialized delegates at the appropriate points. Moreover, the AO mechanism of *advice* introduction may be unfamiliar to many developers, as AO techniques are still relatively "new" and are not yet broadly taught in typical Computer Science or Computer Engineering programs.

Thus, the question of which technique is ultimately "better" depends on the requirements of the system, the goals of the designer and the skills of the development team, which are difficult to quantify in the general case. Nevertheless, there are interesting and measurable differences, even in highly simplistic metrics, that shed some light on the relative merits of these two techniques.

Basic Metrics

Once the refactored versions of the Traffic Estimator were complete, a handful of very basic metrics were applied to get a first impression of the relative sizes and complexities of each version. These metrics, the results for which are presented in Table 5.2, are:

- **File Count (Files)**, which is simply the number of files that comprise a given version of the artifact;
- **Vocabulary Size (VS)**, which is a count of the number of classes and/or aspects that comprise the artifact;
- **Source Lines of Code (SLOC)**, which is a count of the functional (that is: excluding comments and blank lines/whitespace) lines of code in the artifact, as determined by the open-source *sloccount*[61] tool.

Technique	Files	VS	SLOC
Direct Encoding (DE)	2	1	580
Aspect Oriented (AO)	7	6	657
Object Oriented (OO)	11[5]	8[5]	833

Table 5.2: Traffic Estimator: Basic Software Metrics

As is expected, both the AO and OO approaches span many more files and classes/aspects than the directly-encoded version. This is partly the point of the new approaches: to pull the supplemental effects out of the core implementation and isolate them in separate units of encapsulation. However, more is not necessarily better, and other analyses, such as the Concern Diffusion metrics discussed in the next chapter, can provide additional insight along these lines. In terms of lines of code, it is commonly understood that modularity usually comes at the expense of brevity (and sometimes efficiency), so it is not surprising to see a marked increase in the amount of code necessary to implement the AO and OO versions as opposed to the DE version. In this case, however, it is noteworthy that the OO approach required a few more files and classes, and significantly more source code, to do its job than the AO approach. This disparity can be traced to two primary causes:

1. It was necessary to define generic utility classes and several additional intermediary classes to allow individual optional effects to be bound to the core OO implementation, where the AO concept of a *pointcut* was a much more natural fit to the problem;
2. There is generally more overhead involved in declaring and implementing a subclass for the OO implementation than there is in the corresponding advice declaration in the AO implementation.

While these basic results may somewhat favor the AO over the OO technique, none of them points to any tangible benefit over the simple, direct encoding of the effects within the core algorithm. Still, an overhead of 15-30% in size is not prohibitive, and it remains relatively stable through the other two modules.

[5] 2 files and 1 class in the Object-Oriented version are support components that were generated for this first refactoring, but will be shared among other refactored components

5.3 Precedence Estimator

The second behavioral component to be refactored was the Precedence Estimator, which was responsible for monitoring the upcoming intersection for:

- The **precedence ordering** among vehicles stopped at the various stop lines that comprise the intersection;
- Whether the intersection is **clear of other traffic**, which was a prerequisite for proceeding through the intersection, per the Urban Challenge rules[16];
- Whether Boss must **yield** to any traffic that has right-of-way, such as when merging into traffic at a T-intersection.

As with the Traffic Estimator, the role of the Precedence Estimator was defined in terms of two comparatively simple output Subjects, *HavePrecedence* and *IntersectionIsClear*, that were used by the downstream Transition Manager to gate the transmission of goals to the motion planning subsystem. This arrangement of Subjects, including the full range of input data that was used by the Precedence Estimator, is shown in Figure 5.8.

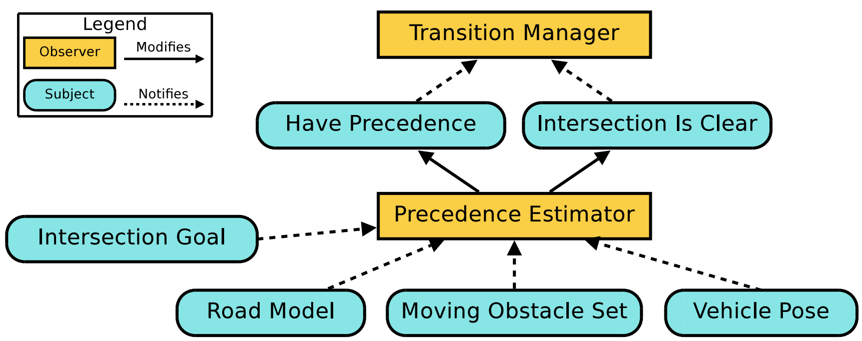


Figure 5.8: Precedence Estimator collaboration diagram, showing critical input and output Subjects according to the Observer[21] pattern.

The algorithms behind the output Subjects in Figure 5.8 are much more complex than those in the Traffic Estimator, due in large part to the complexity of the rules governing intersections and the application of various timeouts to resolve deadlocks and other aberrant situations. In particular, the Urban Challenge rules[16] specified that if there is no activity at an intersection for ten seconds, then the robot should proceed through the intersection regardless of estimated precedence or clearance. While the ten-second timeout was easy enough to implement, the estimation of intersection "quiescence" was a more delicate process that included becoming incrementally more aggressive about culling irrelevant obstacles, while simultaneously guaranteeing that Boss would not charge into an "obviously" busy intersection.

The Precedence Estimator, whose algorithms are more thoroughly described in [3], solves the intersection handling problem by deriving a collection of "occupancy zones": one for each place that a vehicle would have to wait for precedence, one for each lane where traffic passes through the intersection without stopping, and one for the bounding box of the intersection itself. These are updated when the "Intersection Goal" in Figure 5.8 changes, and they provide a geometric description of where all relevant traffic might be for the upcoming intersection.

As with the Traffic Estimator, updates to the list of moving obstacles lead to an iteration over each such obstacle, testing it for relevance to each context, and, if relevant, incorporating it into the associated calculations for precedence, yields and clearance. Baseline relevance is determined in terms of the *primary* data in the `MovingObstacle` representation, namely the pose and size of the candidate, which is used to derive a bounding polygon that is tested for overlap with the various occupancy zones mentioned above. The associated calculations, such as computing the estimated time of arrival (ETA) for traffic in yield lanes, also depends only on primary data, including pose, size and the velocity vector.

In terms of supplemental data, two properties, `isObservedMoving` and `laneAssociations`, have effects in the Precedence Estimator. However, the complexity of the effects and their interaction with the core algorithms posed new and interesting challenges while refactoring according to the proposed techniques. In particular, as Boss was limited to traffic detection via LASER and RADAR sensors, it was prone to spuriously identifying roadside debris and vegetation[6] as candidate vehicles, especially within the comparatively unstructured confines of an intersection. Discrimination between these false positives and "true", or at least "worth treating as true", traffic fell to the supplemental data mentioned above, whose effects, again rephrased as functional requirements are:

PE.S.1 A moving obstacle must be associated with at least one lane, as indicated by its `laneAssociations` listing, in order to be considered as a candidate for precedence among stop-lines, occupancy of the intersection, or as moving traffic for yield computations. This will reduce the impact of false-positives caused by vegetation or roadside debris.

PE.S.2 Once the timeout for overriding intersection clearance has expired, a given obstacle must have its `isObservedMoving` property set to *true* in order to be considered as a candidate for any computations in the Precedence Estimator. This will reduce the impact of false-positives caused by vegetation or roadside debris, both in the intersection and at stop-lines, that may be otherwise blocking the forward progress of the robot.

PE.S.3 A given obstacle must have its `isObservedMoving` property set to *true* to be considered for yield computations. This will cause the robot to more aggressively cull candidate obstacles, reducing the impact of false positives without causing the robot to completely ignore highly-probable traffic in or approaching the intersection. This effect shall be run-time configurable (e.g. enabled/disabled through the TROCS[58] utility) to allow in-situ performance evaluation.

PE.S.4 The intersection override timeout shall be disabled whenever there are any obstacles in the intersection with their `isObservedMoving` properties set to *true*. This will prevent Boss from charging blindly into an otherwise busy intersection, but will still cull probable false positives caused by vegetation or other perception artifacts within the intersection.

[6] A common and sufficiently frustrating phenomenon that these were dubbed "veggie-cars" by the team.

It was immediately clear that these require more than two comparatively simple points of variation, as was the case with the Traffic Estimator, and that the effects would be more coupled to the inner workings of the core algorithm. For instance, **PE.S.2** depends on the state of the "intersection override" timer, which is otherwise internal to the Precedence Estimator. In addition, **PE.S.3** introduces the first instance of run-time configurability as part of a supplemental effect, requiring any implementation thereof to "know" a great deal more about the overall system, such as the specific usage of the Observer pattern to allow interactive modification of Subjects. Lastly, the implementation of **PE.S.4** requires more extensive knowledge of the overall flow of the Precedence Estimator's algorithm in order to cache the notion of "observed moving vehicles are in the intersection" and bind that to the idea of whether or not the intersection is "quiescent" at some other point in the algorithm. These issues are further compounded by the presence of an additional requirement that, while it depends only on *primary* data, interacts curiously with the first three supplemental effects:

PE.C.1 Under no circumstance shall a candidate obstacle whose speed exceeds a certain threshold be ignored for any purposes within the Precedence Estimator. The value of this "maximum ignorable" speed shall be load-time configurable.

Because this otherwise *core* requirement must be able to override any and all *supplemental* effects, it represented a bizarre inversion of design goals. In this case, the best course of action was to introduce this core effect as though it were supplemental, using the same interfaces (delegation or XPI) and enforcing an order of application that ensures that it always has priority over all others. In other words, the requirement for a "maximum ignorable" speed acknowledges the probable existence of supplemental effects based on other data and preempts their functionality, so it is only natural that the most effective approach masquerades as an supplemental effect. This introduces another gray area in the segmentation of robotic algorithms into core and supplemental effects that is left to the developer's intuition, but it also demonstrates a strength of the proposed approach in that it easily accommodates such functionality.

Redesign Experiments

As with the Traffic Estimator, the Precedence Estimator was originally implemented in a single class that made no specific effort to isolate the supplemental effects listed above. In contrast, however, the exposition of variability in the core algorithm was more delicate, leading to new and interesting patterns in each of the OO and AO designs.

Object-Oriented Design

The application of the five effects listed above spanned six distinct points in the core Precedence Estimation algorithm. Rather than exposing them each in individual delegates, as was done for the Traffic Estimator, these points of variability were grouped into three delegation interfaces:

- **ObstacleUpdateProcessDelegate**, which exposes the critical stages in the overall processing of updated obstacle lists, specifically the beginning thereof, as *obstacleSe-*

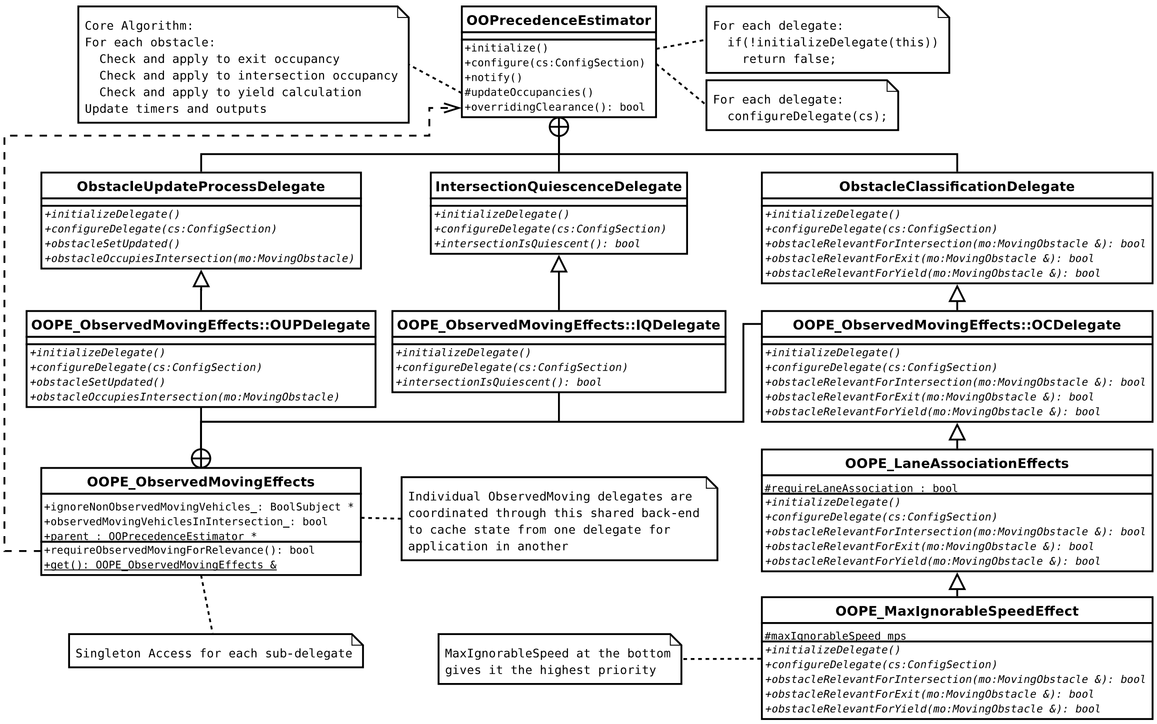


Figure 5.9: Object-Oriented redesign of the Precedence Estimator, isolating supplemental effects in separate delegate classes.

tUpdated(), and the point at which a vehicle is identified as occupying the intersection, as *obstacleOccupiesIntersection()*[7], pursuant to **PE.S.4**.

- **IntersectionQuiescenceTestDelegate**, which also supports **PE.S.4**, exposes the test for whether the intersection is "not busy", which in turn activates the override timeout mentioned above.

- **ObstacleClassificationDelegate**, which reflects the similarity between the tests underlying **PE.S.1**, **PE.S.2**, **PE.S.3**, and **PE.C.1** by collecting the relevance tests for exits, yield lanes, and the intersection itself into a single delegation interface.

This grouping allowed for a reduction in the number of different delegate classes to be managed, but at the cost of possible priority conflicts between the various relevance tests. As with the Traffic Estimator's speed estimation delegate, however, the effects on the relevance tests were largely the same, i.e. the candidate obstacle would be considered "irrelevant" for one reason or another, so the order was not important. The exception to this is the implementation of **PE.C.1**, which required that the corresponding delegate, `OOPE_MaxIgnorableSpeedEffect`, be placed at the bottom of the inheritance chain. This imposed no special costs on the design, however, as the priority ordering already had to

[7]For increased generality, as will be discussed relative to the change experiments in Chapter 8, it would have been worthwhile to also expose the points where obstacles occupy exits and yield lanes, but this initial refactoring focused on the minimal changes necessary so as not to overly bias any metric results.

be explicitly composed, as discussed above. In this way, the application of **PE.S.1** and **PE.C.1** was straightforward.

The implementation of the other effects, however, required some mechanism for sharing state between multiple delegate classes, which was achieved with a Singleton[21] container class, `OOPE_ObservedMovingEffects`, which:

1. Coordinates the shared state between the two delegates that implement **PE.S.4**, the nested `OUPDelegate` and `IQDelegate`, and

2. Manages the run-time configurability and internal timeout dependencies on behalf of the nested `OCDelegate`.

The extra dependencies introduced to handle effects that span multiple delegates lead to a more complicated design, negatively impacting some of the more advanced metrics discussed in Chapters 6 and 7. This reflects one of the limitations of OO methodology in this context: that the flexibility of having many separate delegation interfaces must be weighed against the problem of dealing with effects that span multiple such interfaces. The AO design, due to the nature of aspects, need not make such a tradeoff, which once again yields a more concise design.

Aspect-Oriented Design

The AO redesign of the Precedence Estimator, shown in Figure 5.10, follows the same patterns of benefit and drawback as the Traffic Estimator. That is, the description of core algorithmic variability in `XPI_AOPrecedenceEstimator` is more concise and flexible than the OO counterparts, but at the cost of requiring several trivial methods to be factored out of the core implementation, such as *obstacleIsRelevantForIntersections()*. The "core" implementation of this is method is to simply return *true*, and special comments are put in place to convey "this is specifically meant for AO advice introduction".

In addition, the effects of individual supplemental data are easily and concisely described in separate aspects as *around* or *after* advice against pointcuts in the XPI, with none of the extra dependencies necessary to bridge points of variability in the OO design. This again points toward a more natural fit of AO techniques to the problem of supplemental effect introduction.

There are, however, two critical differences between this AO design and that of the `AOTrafficEstimator`. First, in order to place itself at a higher priority than other effects, `AOPE_MaxIgnorableSpeedEffect` must "know" about all other aspects that affect the "relevance" pointcuts in the XPI. This "knowing" erodes the benefits of "obliviousness", and, in the limit, can lead back to the explicit listing of aspect precedence for all join points, as is the default case with the composition of OO delegates discussed above. Still, this approximation of one of the drawbacks to the OO design is a worst-case scenario, as the AO mechanism is much more flexible, allowing for the full span of complete, to partial orderings between aspects.

Second, in order to accommodate runtime configurability for **PE.S.3**, and to interact correctly with the intersection timeout for **PE.S.4**, `AOPE_ObservedMovingEffects` must look "past" or "around" the XPI to acquire direct dependency on the

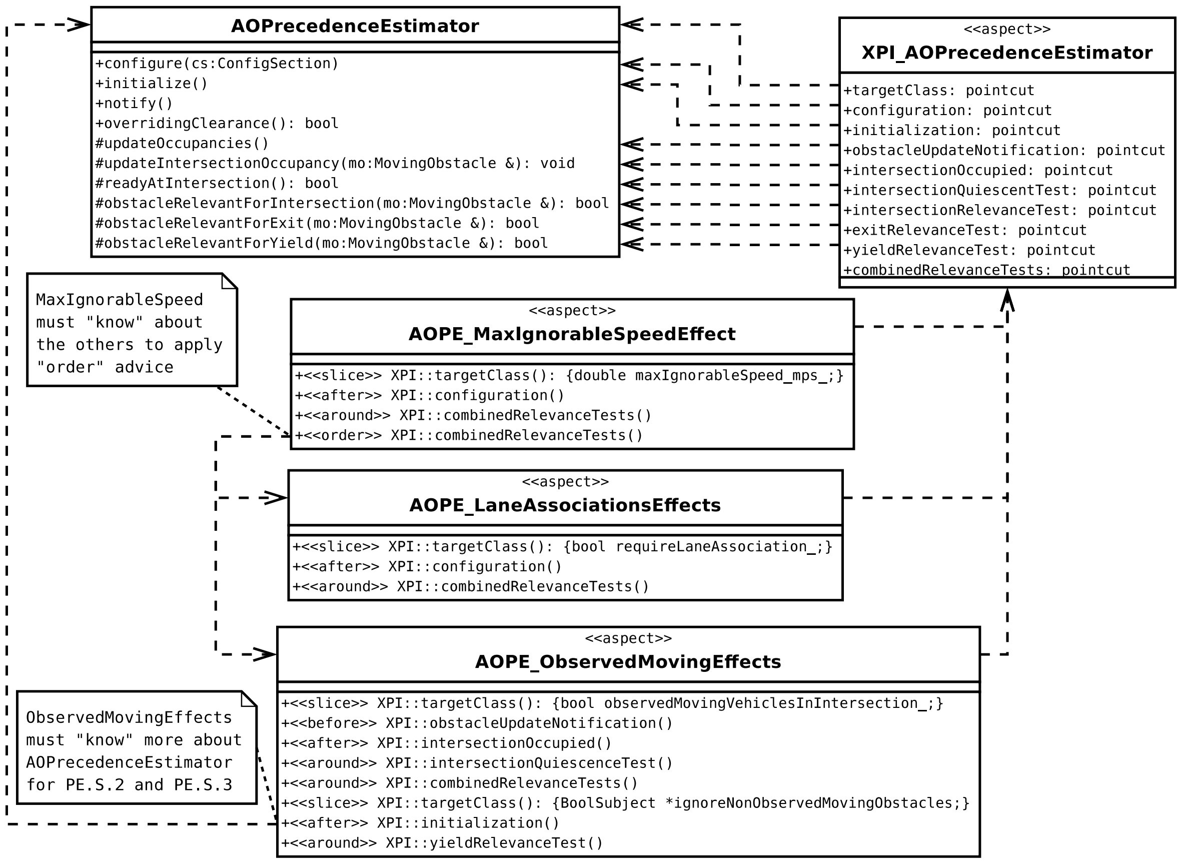


Figure 5.10: Aspect-Oriented redesign of the Precedence Estimator, isolating supplemental effects in separate aspects through a crosscutting programming interface.

`AOPrecedenceEstimator` implementation. While such dependencies are clearly undesirable, the OO design exhibited the same problem, which points toward some amount of underlying complexity that cannot be "designed away". This may, in turn, indicate that some restructuring of the core algorithm, such to introduce a more generic mechanism for run-time configurability[8], and perhaps to refactor the specific structure and usage of the *intersectionIsQuiescent()* method, may be warranted in order to better accommodate supplemental effects. However, as with the identification of "likely" points of variability beyond those necessary to express existing supplemental effects, such modifications were beyond the scope of the initial, minimalist refactoring. The issues of identifying likely points of variation *a priori* and of structuring core algorithms to accommodate supplemental effects are discussed in Chapter 8, where several candidate extensions to novel input data are considered for these three software components.

[8]As-built, the behavioral subsystem relied on individual *Subjects* that were mutated by a dedicated *Observer* called the "Diagnostic Interface"[4] to provide run-time configurability. This meant that supplemental effects that required run-time configurability had to "know" about the specific usage of the Observer pattern, along with their core algorithm's place therein, in order to implement runtime configurability. The presence somewhat more generic way to register for change-notifications for configuration variables would reduce the complexity of both the AO and OO designs for the Precedence Estimator.

Basic Metrics

The same basic metrics applied to the Traffic Estimator were applied to the Precedence Estimator, with the results summarized in Table 5.3:

Technique	Files	VS	SLOC
Direct Encoding (DE)	2	1	1593
Aspect Oriented (AO)	7	6	1782
Object Oriented (OO)	11[9]	10[9]	2055

Table 5.3: Precedence Estimator: Basic Software Metrics

For these basic metrics, the results are strikingly similar in terms of file count and vocabulary size to those for the Traffic Estimator. In terms of relative sizes, this is also the case for source lines of code, with the AO implementation roughly 15% larger than the DE implementation, and the OO implementation 15% larger, in turn, than the AO implementation. While this clearly depends on the complexity of the core algorithm and the number and complexity of optional effects, this may still provide a useful rule-of-thumb for gauging the relative initial development costs for each design variant.

5.4 Merge Planner

The final component to be refactored was the Merge Planner, which was responsible for identifying, synchronizing with, and ultimately merging into gaps between traffic in an adjacent lane. This target, or "intended" lane is determined by a separate component, the Lane Selector, in a separation of responsibilities analogous to the *meta-tactical* and *tactical* levels in the simulated highway lane-planning work described in [36]. This relationship is illustrated in Figure 5.11, showing the Lane Selector's responsibility of determining both the "current" and "intended" travel lanes, and the Merge Planner's following role of determining which of those two lanes to "command", along with how fast the robot should be travelling in order to synchronize with the best merge gap.

The role of the Merge Planner is much more complicated and dynamic than those of the Traffic Estimator, which focuses on single-lane travel, and the Precedence Estimator, which performs its work while the vehicle is stopped, awaiting the appropriate time to proceed through an intersection. The Merge Planner, in contrast, must consider moving traffic in two separate lanes and perform its reasoning while the robot is in motion, with the ultimate output triggering potentially dangerous lane changes between two other vehicles. As such, the Merge Planner is by far the largest component considered in this study, being nearly twice as large (in lines of code) as the Precedence Estimator.

The development of the Merge Planner also differed from the other two components in three critical ways:

1. Where the first two components were developed by the author of this thesis, the Merge Planner was instead developed by another person, with different training, experience

[9] 2 files and 1 class in the Object-Oriented version are the support components that were initially created for the Traffic Estimator.

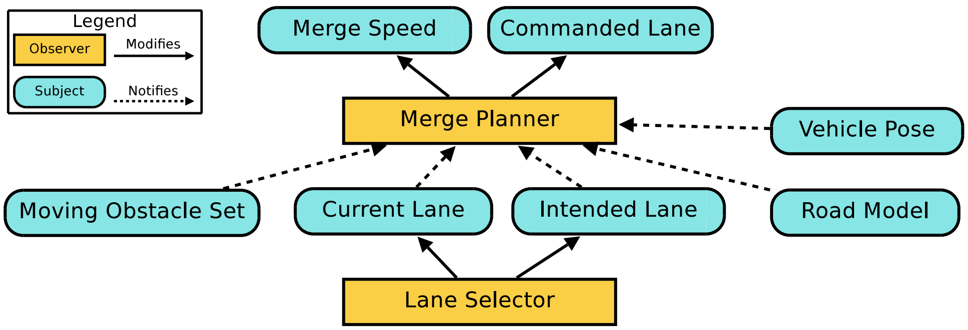


Figure 5.11: Merge Planner collaboration diagram, showing critical input and output Subjects according to the Observer[21] pattern.

and programming style;

2. The Merge Planner was one of the last significant functional elements to be completed for the Urban Challenge, which meant that its development was under much more intense time pressure and was not revisited or refined as were the first two components, leaving it in a more prototypical state;

3. Relatedly, the perception capabilities of the robot were more reliable and mature, meaning that it was not subject to as many instances of *additional* or *altered* data as the previous two.

Despite these differences, the Merge Planner's algorithm was subject to several supplemental effects, many of which are familiar from the previous two components, plus a few others that represent a unique and interesting twist on the supplemental theme:

MP.S.1 Instead of the default geometric checks for whether an obstacle is in Boss's current lane of travel, the Merge Planner shall check the `laneAssociations` list for the maximally-probable lane, and use that as the obstacle's lane of travel.

MP.S.2 Distant obstacles with dubious historical motion, as indicated by the `isObservedMoving` property being set to *false*, shall be excluded from merge calculations as probable RADAR false positives by imposing a shorter culling range than the default. This alternate range shall be configurable, with a default set to $39m$, corresponding to the effective range of high-fidelity laser scanners on Boss.

MP.S.3 Regardless of the instantaneous velocity of a candidate obstacle, it shall be treated as though it is stopped at its current location unless its `isMoving` property is set to *true*

MP.S.4 Regardless of the instantaneous velocity of a candidate obstacle, it shall be treated as though it is stopped at its current location unless its `isObservedMoving` property is set to *true*

MP.S.5 When merging in front of an obstacle with dubious instantaneous motion, as indicated by the `isMoving` property being set to *false*, that obstacle shall *not* be afforded the minimum safety gap of one vehicle-length, reflecting the likelihood that the obstacle is either a stalled vehicle or a non-vehicle road blockage.

MP.S.6 When merging in front of an obstacle with dubious historical motion, as indicated by the `isObservedMoving` property being set to *false*, that obstacle shall *not* be afforded the minimum safety gap of one vehicle-length, reflecting the likelihood that the obstacle is either a stalled vehicle or a non-vehicle road blockage.

MP.S.7 Regardless of the instantaneous velocity of a candidate obstacle, it shall not be considered to be "oncoming" traffic unless its `isMoving` property is set to *true*

MP.S.8 Regardless of the instantaneous velocity of a candidate obstacle, it shall not be considered to be "oncoming" traffic unless its `isObservedMoving` property is set to *true*

The first two effects were familiar from the previously discussed components, where **MP.S.1** suppresses underlying geometric calculations similar to **TE.S.5**, and **MP.S.2** imposes an alternate threshold similar to **PE.C.1**. The remaining effects, **MP.S.3** through **MP.S.8**, brought a new challenge to this work. The careful reader will note that these are paired such that `isMoving` and `isObservedMoving` have identical effects in three different places in the Merge Planner's algorithm. To place these in the proper context, it is necessary to discuss the original implementation in somewhat more detail.

Due to a confluence of complexity and stylistic issues outlined above, the Merge Planner adopted a "batch processing" approach, where the input data was passed through several stages of computation in a pattern not unlike the pipe-and-filter paradigm[49] commonly used in image processing, large-scale simulation, and the Unix shell environment. At each stage of computation, the data is "transformed" from one type to the next, with the number of intermediate types dependant on the number of processing stages. The Merge Planner made use of three such stages, whose intermediate data types were declared as internal, or "nested" utility classes, and are shown in Figure 5.12

Tracing the path of `MovingObstacle` data through the Merge Planner yields an important view of the processing that takes place therein, especially exposing critical details to the application of **MP.S.3** through **MP.S.8** above. The three places where these six effects are applied are found at three different stages of the processing pipeline shown in Figure 5.13.

The application of these effects at places where the raw `MovingObstacle` contents were not available meant that the values of `isMoving` and `isObservedMoving` had to be cached and propagated through each of the three intermediate types shown in Figure 5.12. Rather than deal with them separately, their composed value,`(isMoving && isObservedMoving)`, was cached and propagated through these types, yielding the three sets of identical requirements above. The introduction, management, and application of this *derived-supplemental* data imposed several additional requirements, the first three of which pertain to augmentation of the intermediate types to be able to hold the necessary data:

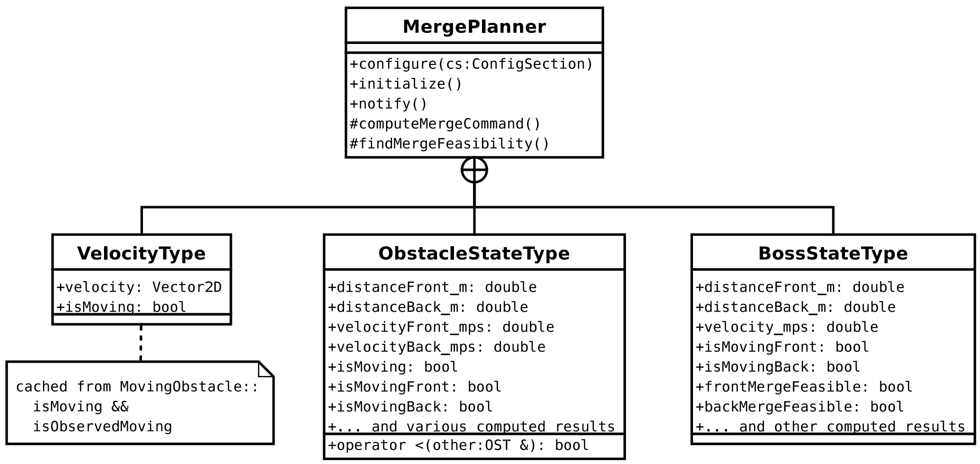


Figure 5.12: Original design for the Merge Planner, showing intermediate obstacle types used in the merge feasibility calculations.

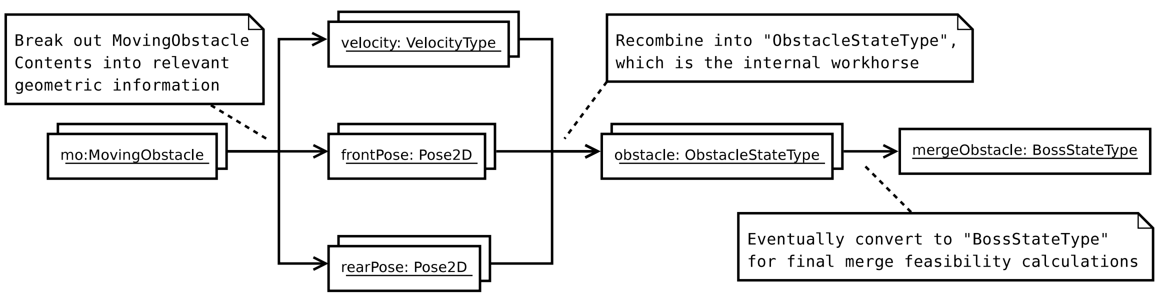


Figure 5.13: Data flow diagram for the Merge Planner, showing propagation of `MovingObstacle` data through the three intermediate types in Figure 5.12.

MP.D.1 The intermediate `VelocityType` class shall be augmented with a boolean member, `isMoving`, pursuant to the application of effects **MP.S.3** through **MP.S.8**.

MP.D.2 The intermediate `ObstacleStateType` class shall be augmented with three boolean members: `isMoving`, `isMovingBack`, and `isMovingFront`, pursuant to the application of effects **MP.S.5** through **MP.S.8**.

MP.D.3 The intermediate `BossStateType` class shall be augmented with two boolean members: `isMovingBack`, and `isMovingFront`, pursuant to the application of effects **MP.S.7** and **MP.S.8**.

The next six requirements have to do with preserving semantics when deriving new obstacle instances from the contents of existing instances:

MP.D.4 When populating a `VelocityType` instance from the contents of a `MovingObstacle` instance, the value of `VelocityType::isMoving` shall be set to true if the `MovingObstacle::velocity` vector is nonzero.

MP.D.5 When populating a `VelocityType` instance from the contents of a `MovingObstacle` instance, the value of `VelocityType::isMoving` shall only be *true* if the corresponding value of `MovingObstacle::isMoving` is also *true*

MP.D.6 When populating a `VelocityType` instance from the contents of a `MovingObstacle` instance, the value of `VelocityType::isMoving` shall only be *true* if the corresponding value of `MovingObstacle::isObservedMoving` is also true.

MP.D.7 When populating an `ObstacleStateType` instance from the contents of a `VelocityType` instance, the value of `VelocityType::isMoving` shall be propagated to all three of the `isMoving{Front,Back}` properties introduced in **MP.D.2**.

MP.D.8 When combining two `ObstacleStateType` instances to form a single, larger merge obstacle, the `isMovingFront` value shall be retained from the "front" obstacle, the `isMovingBack` value shall be retained from the "rear" obstacle, and the `isMoving` property of the resulting instance shall be the bitwise-or of `isMoving` properties of the two contributing instances.

MP.D.9 When populating a `BossStateType` instance from the contents of an `ObstacleStateType` instance, the values of `isMovingFront` and `isMovingBack` shall be propagated to the homonymous members introduced in **MP.D.3**.

The last three derived-supplemental effects represent the actual application of the ordinal requirements, **MP.S.3** through **MP.S.8**, via the cached intermediate values for `isMoving`, `isMovingFront`, and `isMovingBack`:

MP.D.10 When populating a `VelocityType` instance from the contents of a `MovingObstacle` instance, the values of `VelocityType::velocity` and `VelocityType::velocity_STD` shall be overridden to zero if the value of the corresponding `VelocityType::isMoving` field, introduced in **MP.D.1**, is *false*. This fulfills **MP.S.3** and **MP.S.4**.

MP.D.11 When considering a merge maneuver in front of an aggregated merge obstacle, that obstacle shall *not* be afforded the minimum safety gap of one vehicle-length unless the corresponding value of `ObstacleStateType::isMoving` is *true*. This fulfills **MP.S.5** and **MP.S.6**.

MP.D.12 Regardless of the reported velocity of an aggregated merge obstacle, it shall not be considered to be "oncoming" traffic unless the corresponding value of `BossStateType::isMovingFront` is *true*. This fulfills **MP.S.7** and **MP.S.8**.

Given time to revisit the core Merge Planner's design, it seems likely that the three-stage processing pipeline in Figure 5.13 would be reduced to a single stage, wherein the input `MovingObstacle` representation would be used to populate a single internal representation of a "merge obstacle", similar to the intermediate `ObstacleStateType`. This representation was the most expressive of the merging problem, as it supported the combination of obstacles that were "too close" to merge between, and included a facility (the less-than operator) for sorting the obstacles in ascending order along a given lane. Nevertheless, the goal of the redesign experiments in this section is to leave the original implementation as close as possible to its as-written state, focusing on the minimal changes necessary to expose points of variability and extract supplemental effects according to the proposed AO and OO techniques.

Moreover, even though the Merge Planner's algorithm might be simplified to eliminate the need for many of the secondary requirements listed above, the issue of translation through intermediate data types is common in advanced robotic software. As will be discussed in Chapter 9, regarding CLARAty, there will be circumstances where a multi-stage processing pipeline is the best approach, and supplemental data introduced at the beginning may have to be explicitly propagated through all intermediate stages to have meaningful effects at the distal end. As such, the Merge Planner's core design was left as-is, in order to understand the implications of long-pipe processing, and as an excellent case study for designing around supplemental effects in other components with similar structures.

Redesign Experiments

The critical challenge for both alternate designs for the Merge Planner was to introduce new fields in internal structures and manage the propagation. Otherwise, the application of the actual supplemental effects, **MP.S.X** listed above, was straightforward and followed the same patterns established for the other two components.

Object-Oriented Design

Given the complexity of the Merge Planner, and the number of effects listed above, it is necessary to split the presentation of the OO design into several separate parts. To begin, Figure 5.14 shows the accommodation of *derived-supplemental* data introduction via mechanism similar to the idea of "mixins" from [51]. Here, a generic member is added to each intermediate structure, called "extension", whose type is determined by composing one or more specific extension classes, such as `ObstacleStateIsMovingExtension`.

As various *derived-supplemental* data are introduced, the type definition ("typedef") for each extension is updated to include the new information as necessary. This was chosen instead of direct use of templates to avoid having to parameterize the `OOMergePlanner` class declaration against an extension type for each of its internal structures. Supplemental delegates, such as those shown in Figures 5.16 and 5.17, would then access the *derived-supplemental* data through the appropriate `extension` member.

In addition to handling intermediate type extensions, the variability in the Merge Planner is exposed through four delegate interface classes, shown in Figure 5.15. Four of them, corresponding to the places where effects **MP.S.1** through **MP.S.8** are to be applied, are comparatively simple and follow a familiar pattern from the work on the previous two components. The fifth, `IntermediateObstacleTypeDelegate`, is the largest delegation generated for this work, and it corresponds to the introduction, translation, and maintenance of the various intermediate "isMoving" data introduced in Figure 5.14.

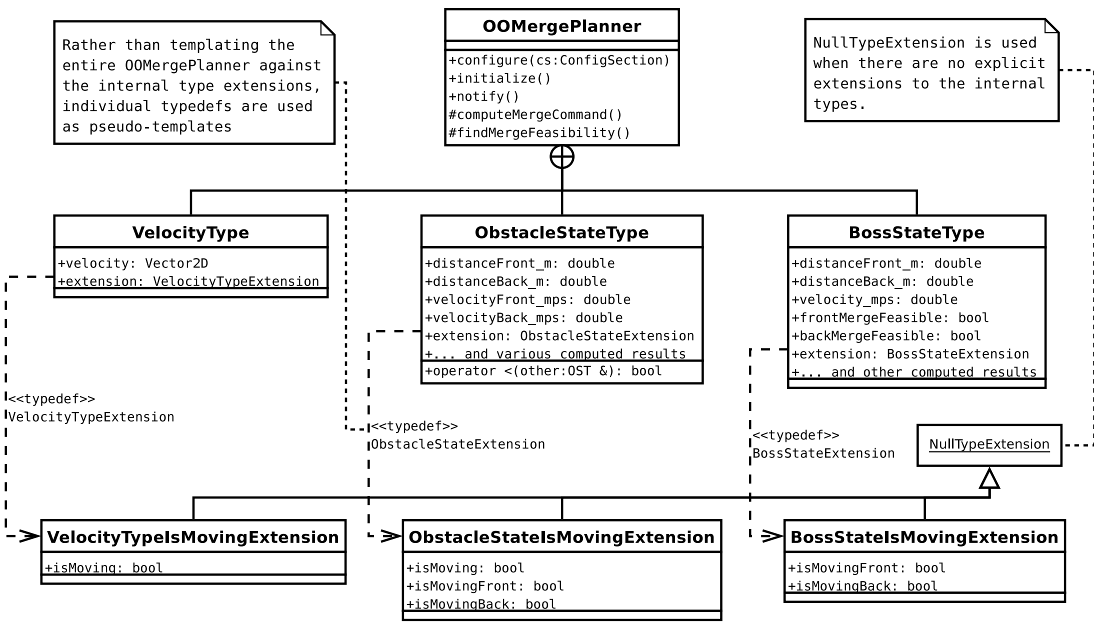


Figure 5.14: Object-Oriented redesign of the Merge Planner, enabling introduction of *derived-supplemental* data into intermediate types via a pseudo-template method.

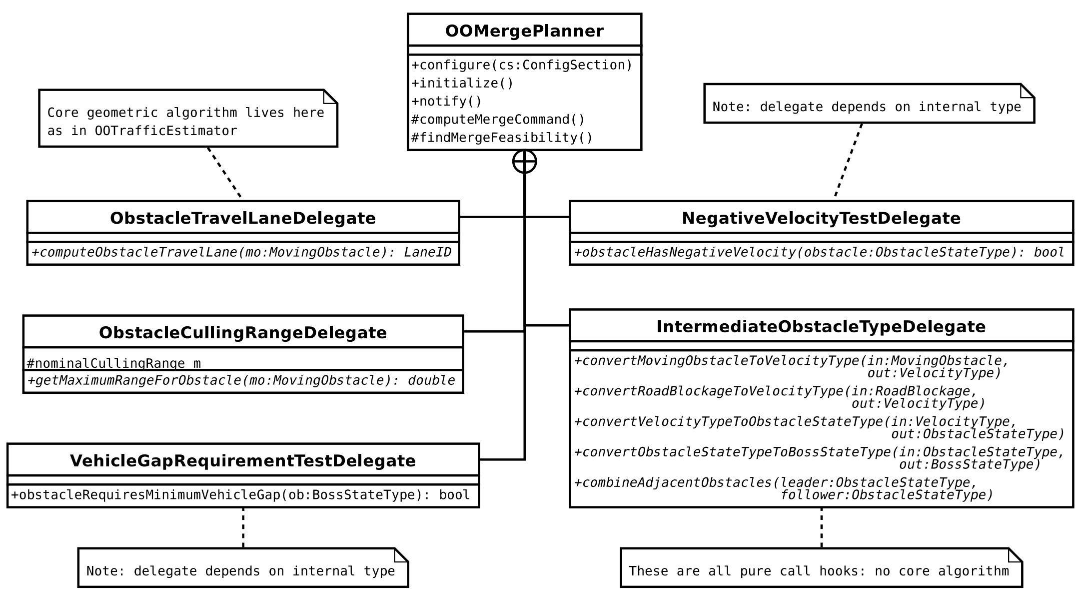


Figure 5.15: Object-Oriented redesign of the Merge Planner, showing exposition of variability in the core algorithm through four delegate classes.

With these delegation interfaces in place, two of the original effects, shown in Figure 5.16, were straightforward to implement, again following established patterns from the Traffic and Precedence Estimators. In fact, the similarity between **MP.S.1** and **TE.S.5** allowed the implementation of the latter to be used as a template, which was particularly useful because

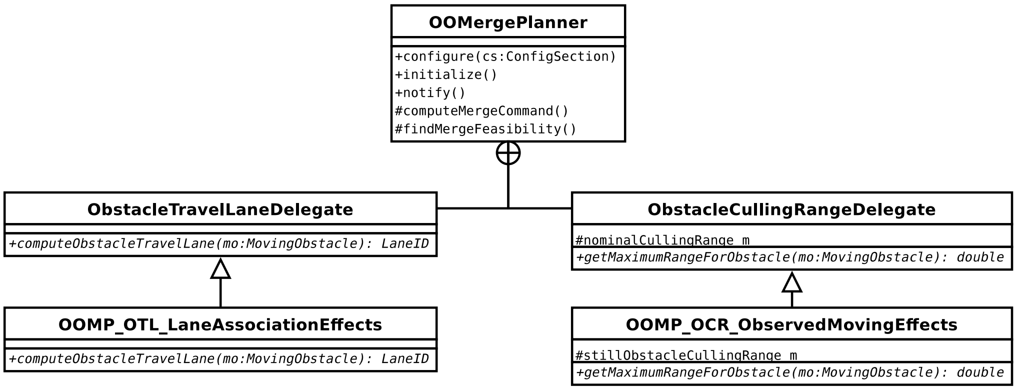


Figure 5.16: Object-Oriented redesign of the Merge Planner, showing the straightforward application of **MP.S.1** and **MP.S.2**.

there was no underlying or "original" implementation for the geometry-only determination of lane occupancy, anywhere in the version history. In this case, the `laneAssociations` datum was already present when the Merge Planner was first developed, so there was never any need for a geometry-only algorithm. If such software were to be used on another system that did not provide the `laneAssociations` datum, there would be no alternative than to develop replacement functionality "from scratch", which is one among the many difficulties engineers face that would be relieved by the methodology proposed in this thesis.

Last, but certainly not least, Figure 5.17 captures the *derived-supplemental* requirements, **MP.D.1** through **MP.D.12**. As mentioned above, with the intermediate "isMoving" properties in place, the application of the actual effects followed the straightforward and familiar pattern first seen in 5.6. In this case, the management and propagation of the intermediate "isMoving" properties was given the highest priority, by placing `OOMP_IntermediateMovingDelegate` at the bottom of the inheritance chain. This ensured that effects that depend on those intermediate data, such as the embodiment of **MP.D.5** in `OOMP_IMD_MovingEffects`, would be seeded with the "correct" initial value.

Aspect-Oriented Design

As with the other components, much of the underlying complexity of the Merge Planner and its supplemental effects are reflected in the AO design as well as the OO design, but the AO design still manages to be cleaner and more concise. This is especially true of the intermediate type introductions, **MP.D.1** through **MP.D.3**, where the AO notion of "class slices" is an exceedingly natural fit. To support this, `XPI_AOMergePlanner` exposes the intermediate types as class pointcuts, such as *intermediateBossStateType*, and `AOMP_IntermediateMovingEffects` applies `<<slice>>` advice to introduce extra data, such in the nested `BossStateSlice`.

The management of these intermediate types does not need to interfere with core functionality, so it is implemented as `<<after>>` advice against the various translation and combination pointcuts in the XPI. This is possible because the propagation of these data through the intermediate types are not *invasive* aspects as discussed in Section 3.2, but are merely *spectative* in that they add "harmless" side effects to the core algorithm. Only when

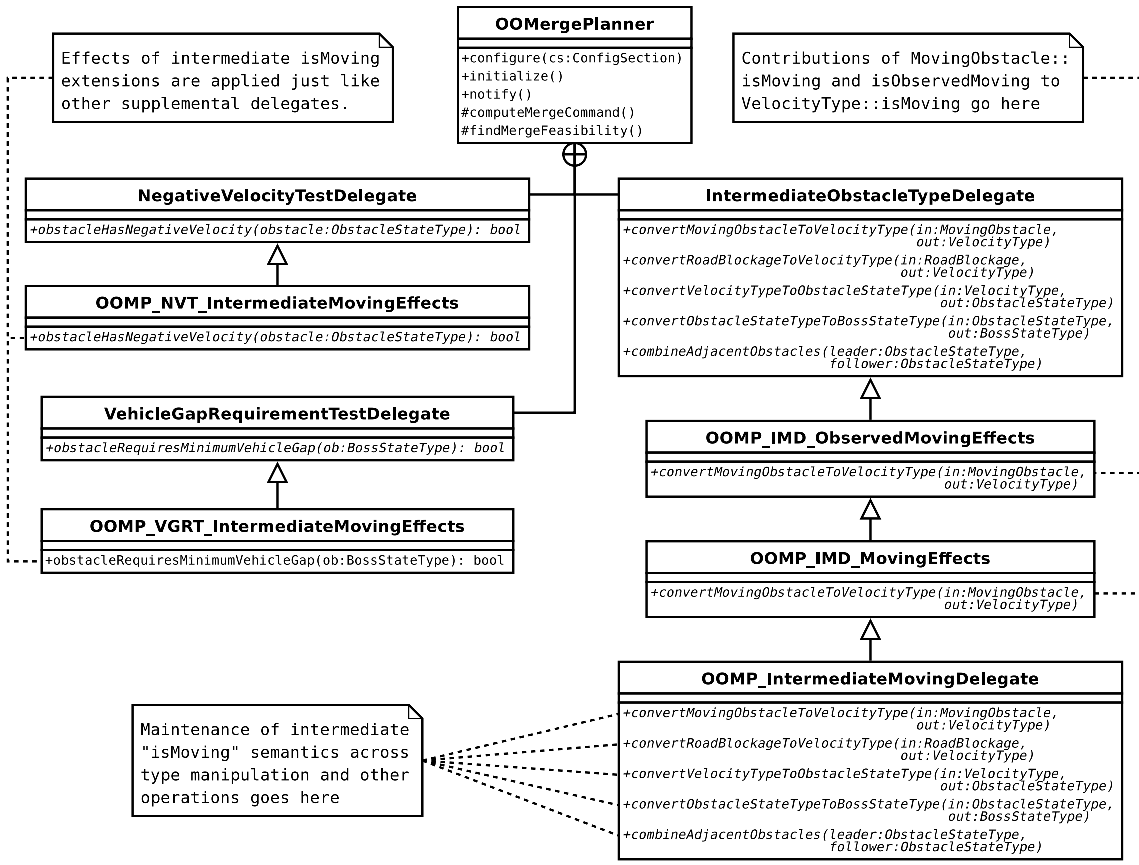


Figure 5.17: Object-Oriented redesign of the Merge Planner, showing the application of derived-supplemental effects **MP.D.1** through **MP.D.12**

the effects of the data are applied at the distal end are they once again *invasive*, and, to the author's knowledge, this usage of *spectative* aspects in support of more *invasive* aspects at "downstream" points in the algorithm has not yet been explored, making this example a noteworthy curiosity contributed by this work.

Otherwise, the only interesting feature of the AO design is that, in order to fulfill **MP.D.5** and **MP.D.6**, `AOMP_MovingEffects` and `AOMP_ObservedMovingEffects` have an implicit dependency on the introduction of `VelocityTypeSlice` onto the *intermediateVelocityType* join-point in the XPI. That is, if the type signature of `VelocityTypeSlice` were to change, or if `AOMP_IntermediateMovingEffects` were to be excluded from the weave, then those two aspects would not function correctly. As with similar "conservation of complexity" discussion regarding the Precedence Estimator, this problem is also present in the OO design, and there is no reasonable way to "design around" it.

Basic Metrics

In terms of the basic metrics applied to the other two components, the results for the Merge Planner, shown in Table 5.4, are consistent with the previous two.

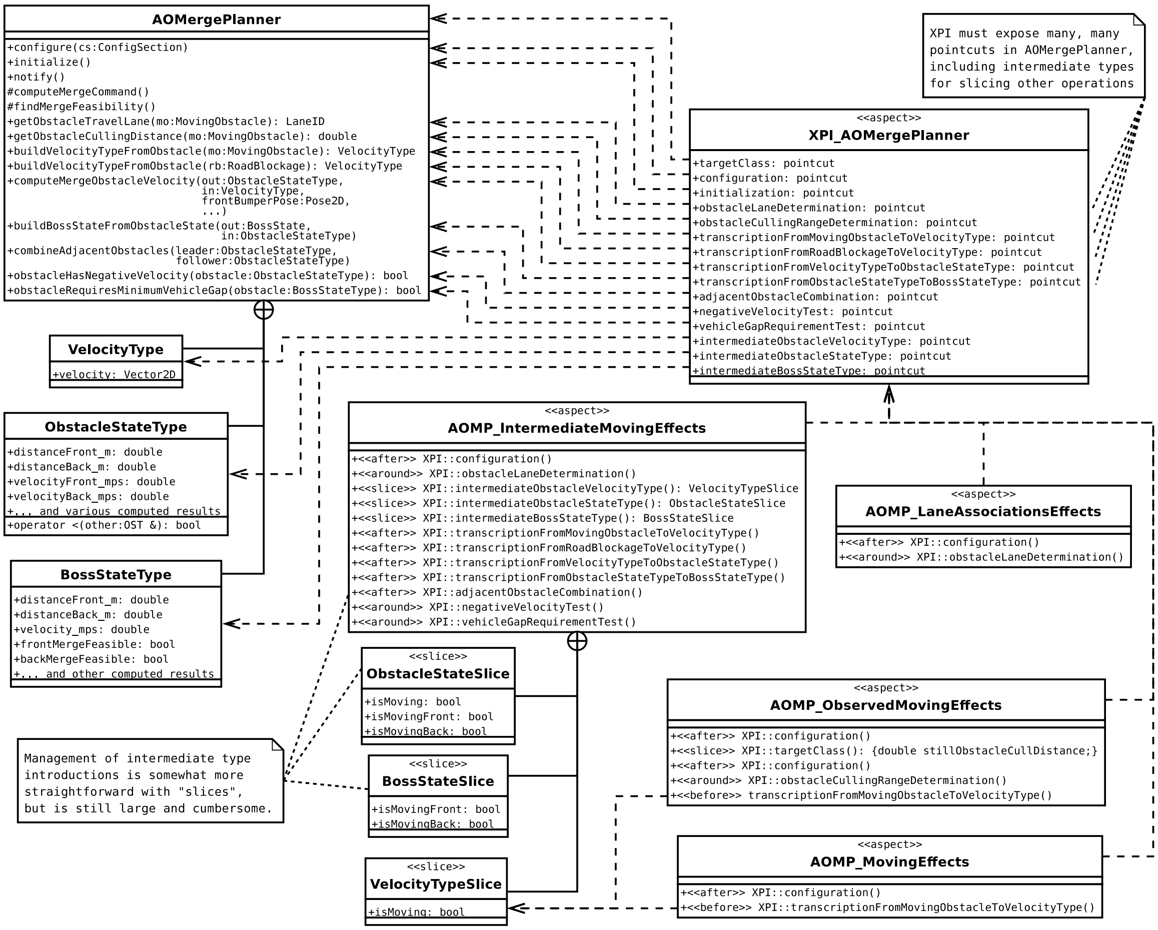


Figure 5.18: Aspect-Oriented redesign of the Merge Planner, isolating supplemental effects in separate aspects through a crosscutting programming interface.

Technique	Files	VS	SLOC
Direct Encoding (DE)	2	6	2965
Aspect Oriented (AO)	7	11	3323
Object Oriented (OO)	17[10]	23[10]	3537

Table 5.4: Merge Planner: Basic Software Metrics

These basic metrics for the Merge Planner show a marked increase in files, classes, and lines of code for both AO vs. DE, and OO vs. AO. The degree of the jump from the OO to AO designs is somewhat more pronounced, due in large part to the extra overhead of intermediate type introductions in the OO technique that were more straightforward in the AO technique. Otherwise, the AO design imposes an overhead close to 10%, and the OO design closer to 20%, which are in line with previous results.

[10] 2 files and 1 class in the Object-Oriented version are the support components that were initially created for the Traffic Estimator.

5.5 Discussion

There are several recurring themes in the supplemental effects described in this chapter that offer useful insights into how the primary vs. supplemental methodology may be applied to other systems. For the most part, these effects represent a binding of a particular supplemental datum to "what it means" in a particular context, such as **MP.S.2**, which can be paraphrased as:

> In the context of the Merge Planner, (`isObservedMoving` = *true*) means that the obstacle can be farther away and still be relevant to merge calculations.

Although each effect is unique to the algorithm to which it is applied, many of them share certain similarities that can guide the identification of supplemental effects in other systems. That is, the majority of the effects listed above interact with the core algorithm in one of three distinct ways:

- Logical contributions to some context-specific relevance test, such as whether an obstacle is "worth considering" for yield calculations (**PE.S.2**),
- Logical contributions to the determination of how, or whether, to incorporate *primary* data, such as how much to "trust" the velocity data (**TE.S.2**), in each application context,
- Substitution of alternate thresholds or offsets, which can be related to "trust" of the *primary* data, but can also pertain to more complicated facets of the core algorithm, such as the determination of obstacle culling range in **MP.S.2**.

Beyond *describing* existing supplemental effects, these categories also allude to the *prescriptive* identification of points of variability in a core algorithm that would be valuable to expose for future adaptation. To a certain extent, these are intrinsically algorithm-specific, and will be difficult to identify *a priori* in the general case. This is discussed in great detail relative to the change experiments in Chapter 8, so it will be sufficient for now to focus on two categories of "support" supplemental effects that pervade the examples in this chapter.

The first such "support" effect is the introduction of load- or run-time configurability, such as the parameterization of a supplemental effect by allowing alternate thresholds or offsets to be specified in a configuration file. To support this, the adaptability interface should permit extensions of the initialization and configuration methods of the core component, and should allow additional member data to be introduced to hold configured parameters in a way that is accessible to the rest of the supplemental effects.

The second "support" effect discussed in this chapter is the introduction and management of *derived-supplemental* data across intermediate data types, such as the **MP.D.X** effects above. Most notably, this requires a mechanism for introducing *additional* data into, or associating such data with, intermediate data types that may be affected by supplemental data. All critical stages in the processing of these intermediate types must be exposed for adaptation, including translation from one type to the next and any operations that derive a new instance from the contents of two or more existing instances. It follows that care should be taken to keep such intermediate types, and the processing path they take through the core algorithm, as simple as possible, thus minimizing the amount of effort required to propagate supplemental data to the places in the algorithm where they are to be applied.

Together, these provide a certain amount of guidance as to how to write adaptable robotic software, and they imply a baseline adaptability interface that is necessary to enable the comparatively simple “support” effects. Relatedly, seeking out these elements in an existing algorithm: tracing the path of input data through intermediate data types, and watching for special-case configuration parameters along the way, is an excellent starting point for identifying pre-existing supplemental effects, along with candidate places where future effects may be applied. This particular theme is explored more thoroughly in the complementary case study in Chapter 9, where the extension of a generic terrain analysis algorithm to accommodate supplemental data from thermal cameras or vegetation detection algorithms also requires augmentation of several intermediate data types to carry the supplemental data to the appropriate application points in the processing pipeline. The derivation and application policies for these derived-supplemental data also create similar opportunities to introduce configurable parameters such as would describe “too hot” to traverse, or “too vegetative” to leave uninspected.

As to the overall merit of the primary vs. supplemental methodology, and the effectiveness of the AO and OO approaches thereto, the basic metrics presented to this point, while relatively easy to measure, focus only on the costs associated with each technique. While these costs do not appear to be prohibitive, and consistently favor the AO design over the OO design, they are not enough in isolation to determine which approach is “better”, or whether the overall methodology actually enhances adaptability. To gain a better understanding of the benefits provided by these techniques, two advanced analysis techniques have been applied to each design of the Traffic Estimator, Precedence Estimator, and Merge Planner, discussed above. The following chapter presents the results from the first such technique, called “concern diffusion analysis”, which quantifies how well the supplemental effects are isolated from the core algorithm, where better isolation typically yields software that is easier to understand and adapt.

Chapter 6

Results: Concern Diffusion

The first set of advanced metrics that have been applied to the refactored urban driving software is drawn from a case study[22] that compares AO implementations of 23 standard design patterns[21] to their stereotypical OO counterparts. To compare the two techniques, this study uses a collection of source code analysis metrics that were shown to be effective quality indicators, both for measuring the intrinsic structure of the artifacts, and for predicting their ability to cope with an assortment of changes that were applied to each pattern. Three of these measurements, the so-called "concern diffusion" metrics, were particularly effective at highlighting differences between the AO and OO designs.

This chapter presents the results of applying these "concern diffusion" metrics to the components discussed in Chapter 5, beginning in Section 6.1 with a discussion of "software concerns" and the three measured types of "diffusion" thereof. The results of applying these metrics to each implementation of the Traffic Estimator, Precedence Estimator and Merge Planner are presented in Sections 6.2, 6.3 and 6.4, respectively. These results are summarized in Section 6.5 before proceeding to Net Option Value analysis in Chapter 7.

6.1 Introduction: Concerns and Diffusion

In the language of software analysis, a "concern" is a conceptually coherent issue that must be resolved as part of developing a software system. The most straightforward concerns take the form of functional software requirements, such as the specification of supplemental effects laid out in the previous chapter. Concerns can also pertain to "non-functional" design and infrastructural issues in a software system, such as message-passing, configuration management, and logging. More abstractly, qualities such as "modularity", "adaptability", and "understandability" are also concerns for a software system, as efforts to enhance such qualities will also affect the structure and usage of the resulting artifact.

The commonality is that all software concerns have some measurable presence in the final implementation of the system, which can consist of source code, compiled executables, configuration files, documentation, etc. It follows that all elements of these artifacts, at all granularities of programs, classes, functions and lines of code, can be traced to one or more concerns for the system. The "diffusion" of any one concern is the degree to which it is *scattered* across the various elements that comprise a given system, and the degree to which the concern is *tangled* with other concerns in the implementation of those elements.

The critical implication is that the more a concern is "diffused", the more difficult it will be to understand for the purposes of adaptation and reuse.

Concerns can also be thought of hierarchically, with a single top-level concern on the order of "the system shall do the right thing", that is recursively decomposed and elaborated upon to the level of detailed functional, non-functional and design requirements. This hierarchical nature has important consequences for the measurement of "diffusion" as discussed above, as the degree of perceived diffusion will depend on the granularity of the concerns that are being measured. At the top of the hierarchy, the aggregated "do the right thing" concern will be *scattered* across the whole system, but will not be *tangled* at all, as there are no other concerns to tangle with.

The goal of a modular design is to provide a strong mapping between implementation artifacts (classes, methods and lines of code) and the increasingly fine-grained concerns, simultaneously reducing the perception of *scattering* as the finest granularity of concerns come into focus while also minimizing the *tangling* of these fine-grained concerns with each other. As discussed in Section 3.2, some "crosscutting" concerns are more difficult than others to isolate and encapsulate, and one of the critical insights in this thesis is that the effects of so-called *supplemental* data might be framed in a similar way, and may benefit from treatment using AO and related techniques.

The "concern diffusion" metrics presented in this chapter provide a quantification of this benefit by directly measuring the amount of scattering and tangling of individual concerns in a software system. The performance of these metrics begins with the enumeration of all the concerns of interest in the artifacts to be tested, followed by a labeling[1] of those artifacts as pertaining to one or more concerns.

In order to accommodate the issues of concern granularity discussed above, the concerns for these artifacts, listed in their entirety in Appendix C, have been specified in a hierarchical fashion, with results computed for each of the three levels of granularity (coarse, intermediate, and fine) therein. At the highest level, there are three "coarse-grained" concerns that describe:

- The implementation of the *core algorithm*, denoted by the prefix **C**;
- The exposure of adaptability, or *variability* (denoted **V**) in the core algorithm;
- The implementation of any *supplemental effects* (denoted **E**).

Core (**C**) and supplemental effect (**E**) concerns will be present in all three designs, as they represent functionality present in the original implementation that must be preserved in any alternate design. The variability (**V**) concerns, on the other hand, map directly to the "extra" code that must be written to accommodate the methodology proposed in this thesis, and are thus only present in the AO and OO designs. Each of these three top-level concerns are broken down at an intermediate level of granularity into:

- Individual stages in the core algorithm, denoted numerically as **C.N**, e.g. "**C.0**: infrastructure, configuration and initialization", which is common to all components;

[1] Even in the presence of industrial strength tools and procedures, this can be a tedious manual process. In this case, of more prototypical development, it required the hand-labeling of nearly 25,000 lines of code.

- Individual adaptation interfaces, also denoted numerically as **V.N**, e.g. "**V.0**: configurability for supplemental effects", which is common to all AO and OO designs;
- The aggregated effects of individual supplemental data, also denoted numerically as **E.N**, e.g. "**E.1**: isMoving effects".

Lastly, the finest level of granularity, which only applies to supplemental effects, maps directly individual requirements listed in Chapter 5. These are denoted numerically as **E.N.M**, e.g. "**E.1.1**: Require isObservedMoving in yield calculations (**PE.S.3**)".

Generally speaking, coarse-grained concerns provide a better overall view, but can also smooth out interesting features in each design. Conversely, the finer-grained concerns can expose these smaller features, but can also be highly sensitive to otherwise irrelevant implementation details. As an example, consider the pseudo-code in Listing 6.1, which extends the example in Listing 5.1 to include labels for core and supplemental effects.

```
C.1  : double TrafficEstimator::estimateObstacleSpeed(
C.1  :                               MovingObstacle &mo) {
C.1  :  double laneSpeed_mps; // computed result for this method
C.1  :
C.1  :   // verify that we "trust" the velocity vector
E.1.1:   if( mo.isMoving &&
E.2.1:       mo.isObservedMoving )
C.1  :   {
C.1  :     // do the "normal" lane speed calculation
C.1  :     // note: this is the "core" algorithm for speed estimation
C.1  :     laneSpeed_mps = projectVelocityVectorOntoLaneHeading();
C.1  :     if(laneSpeed_mps < 0.0)
C.1  :     {
C.1  :       // verify that negative/oncoming velocity is allowed
E.3.1:       if(mo.isPredicted ||
E.4.1:           (oncomingRequiresStrictLaneAssociation_ &&
E.4.1:            mo.laneAssociations.size() != 1)
C.1  :         )
C.1  :       {
C.1  :         // oncoming not allowed: force to zero to be safe
C.1  :         laneSpeed_mps = 0.0;
C.1  :       }
C.1  :     }
C.1  :   } else {
C.1  :     // not trusted: force to zero to be safe
C.1  :     laneSpeed_mps = 0.0;
C.1  :   }
C.1  :
C.1  :   return laneSpeed_mps;
C.1  : }
```

Listing 6.1: Pseudo-code for supplemental effects in Traffic Estimator speed estimation.

Given these labels, three diffusion metrics are computed for each concern by counting:

1. The number of *components*, in this case classes or aspects, that contribute to the concern, called Concern Diffusion over Components (CDC);
2. The number of *operations*, in this case methods or advice directives, that contribute to the concern, called Concern Diffusion over Operations (CDO);
3. The number of times that the concern switches back-and-forth, or *tangles*, with others at the source level, called Concern Diffusion over Lines of Code (CDLOC).

Relative to the contents of Listing 6.1, this implementation would incur one point of CDC for each of the five concerns listed(**C.1**, **E.[1-4].1**), and for each of their parent concerns(**C,E,E.[1-4]**), for their presence in the `TrafficEstimator` class. These would similarly incur one point of CDO for their presence in the `estimateObstacleSpeed` method. Lastly, their CDLOC scores would be computed as follows:

- **C**, **C.1**: 4, for switching "out" at lines (6,15), and "back in" at (8,18)
- **E**: 4, as the complement to **C** above
- **E.1**, **E.1.1**: 2, for "in" at 6 and "out" at 7
- **E.2**, **E.2.1**: 2, for "in" at 7 and "out" at 8
- **E.3**, **E.3.1**: 2, for "in" at 15 and "out" at 16
- **E.4**, **E.4.1**: 2, for "in" at 16 and "out" at 18

It is important to note that the hierarchical grouping of concerns does not necessarily mean that the CDLOC measurement for a coarse-grained concern is a simple summation of the results from its constituent sub-concerns. In fact, this is rarely the case, as shown above, as the aggregated "**E**" concern shows less diffusion (4) than the sum of its constituents (8). This is because the aggregation represents a "bigger picture" look at the diffusion problem, which "smooths over" some concern switches that are only visible at finer granularities.

There is some amount of judgement involved in assigning concerns to some blocks of code, such as the "else" clause at lines 24-27. From a certain perspective, these may be seen as pertaining to both **E.1.1** *and* **E.2.1**, which would incur additional diffusion over lines of code. In this case, however, it is possible to interpret these as the "core" algorithm explicitly including the test for "trust", and the supplemental effects contributing more to the test than the result. Still, there is some ambiguity here, and, as with many other facets of software development and analysis, consistency is more critical than absolute correctness. For this case study, any such ambiguities are resolved by favoring a labeling that results in lower diffusion scores for all designs, yielding consistent, conservative results for each design of each component.

To support analysis at all levels of granularity, the source code has been labeled at the finest level of granularity, and the measurement process has been instrumented to accommodate the naming scheme described above. The results are presented in the following sections as grouped bar graphs to allow side-by-side comparison of the relative scores for each design technique over each specific concern. The bars are colored red for Aspect-Oriented (AO), blue for Object-Oriented (OO), and green for Direct Encoding (DE). The DE design shows negative bars for all “variability” (**V**) concerns as a visual representation that they are irrelevant to that technique. That is, the direct encoding does not attempt to expose points of variability, so those concerns never show up in the hand-labeling of the corresponding source code.

6.2 Traffic Estimator

Concern Diffusion over Components

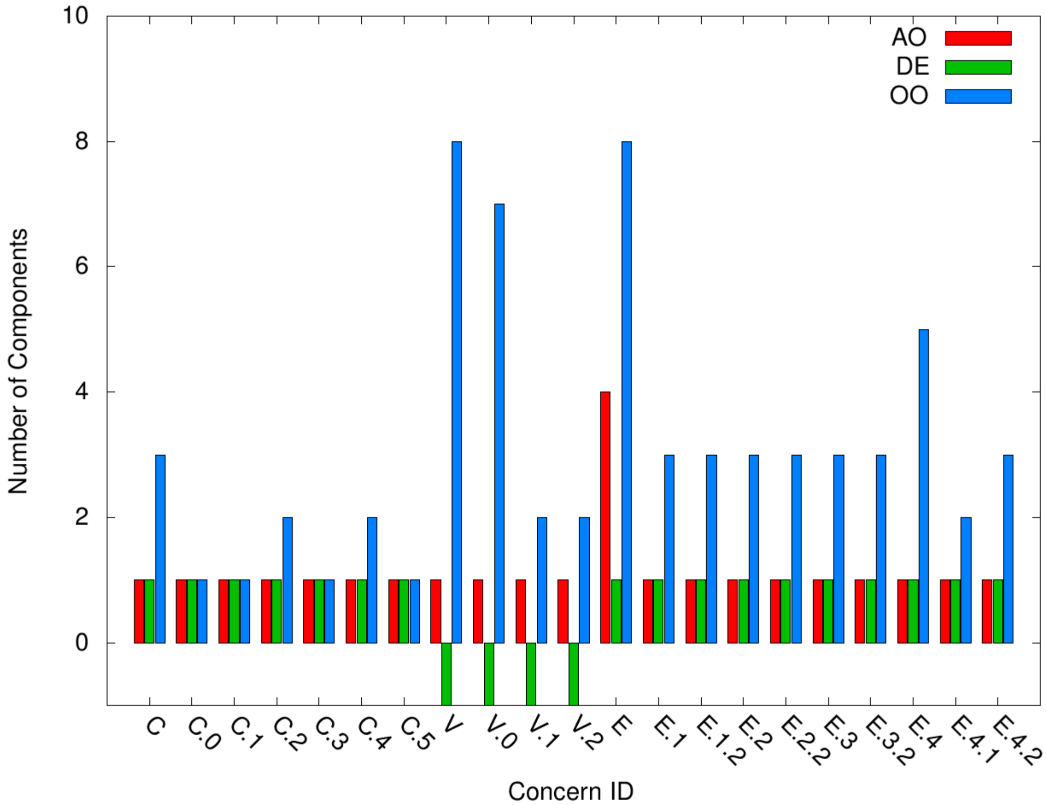


Figure 6.1: Traffic Estimator: Concern Diffusion over Components. See Table C.1 for the full listing of the illustrated concerns.

Figure 6.1 illustrates the CDC scores for the three implementations of the Traffic Estimator. The results are largely consistent with the more simplistic metrics discussed in the previous chapter, i.e., the AO design generally outperforms the OO design, but the diffusion values for each concern reveal several important differences between each of the three designs.

First of all, it is noteworthy that all concerns in the DE implementation, except for the variability concerns as mentioned above, are "diffused" over exactly one component, the original Traffic Estimator class. The AO and OO implementations both show some additional diffusion over components, but a certain amount of this is expected: the point of these two designs is to separate the core and supplemental concerns into individual modules so they can be treated in isolation. However, whereas the AO implementation keeps a one-to-one mapping between the finest-grained concerns and the components that implement them, the OO implementation diffuses some elements of the core algorithm and at least some parts of every supplemental effect over two or three components, reflecting the additional overhead necessary to declare, implement and bind supplemental effects within the limitations of object-oriented methodology.

These limitations are even more pronounced in the center of the graph, which depicts the number of components that take part in exposing the points of variation in the core algorithm. The large disparity between the AO and OO implementations here reflects the more natural way that AO methodology can insert and extend functionality in the core algorithm, by declaring all points of variability in an entirely separate component, the XPI, which is discussed in Section 4.3. The OO implementation, on the other hand, requires separate delegate class declarations per point of variability, and methods on those classes must be invoked at the appropriate places along the core implementation in order to allow the core functionality to be extended or overridden.

Concern Diffusion over Operations

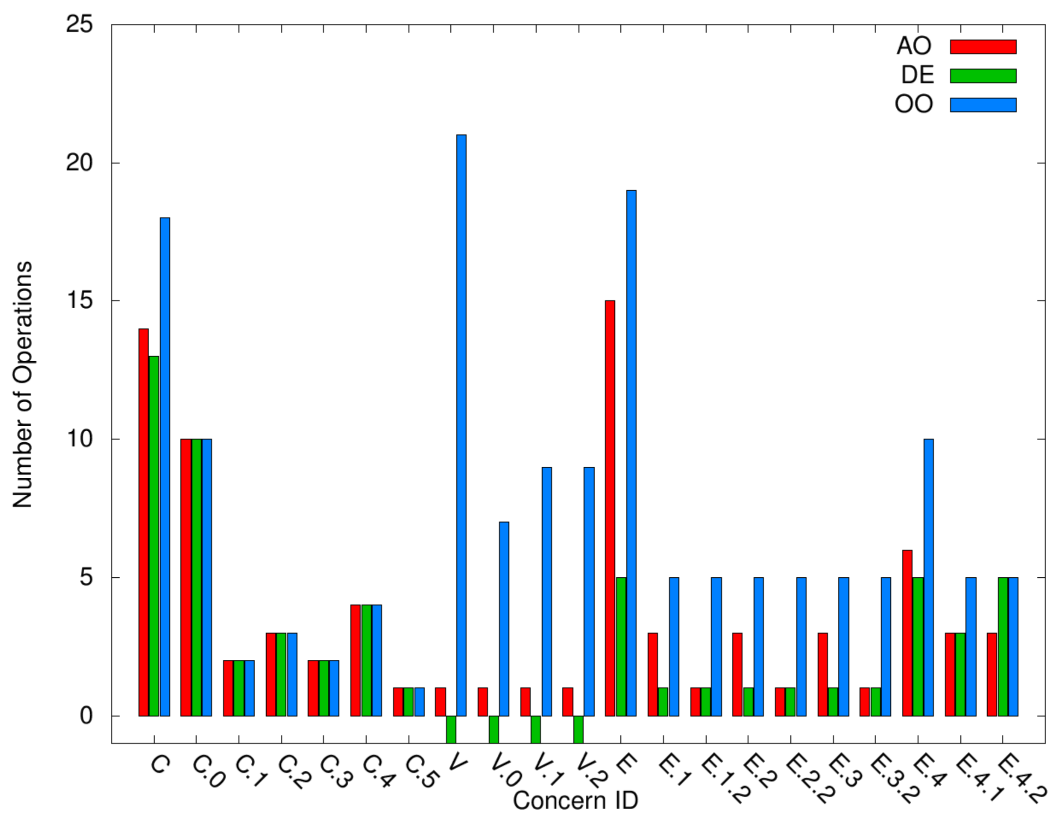


Figure 6.2: Traffic Estimator: Concern Diffusion over Operations. See Table C.1 for the full listing of the illustrated concerns.

The CDO scores for the three Traffic Estimators, shown in Figure 6.2, are largely consistent with the CDC scores, but show more significant variation in the values per individual concern. As with the CDC scores, concerns are generally diffused over more operations in the AO and OO implementations, but this is once again an expected result. The effects on the individual core concerns are minimal, with the increases in the aggregated core concern caused by some basic refactoring of the core algorithm into several additional methods to create the necessary join-points for the AO design, and to provide the option to override the core functionality for the OO design. This highlights the fact that the proposed techniques for applying supplemental effects require some consideration in the design and implementation of the core algorithm. The expected benefit of this additional work is that subsequent variations on the core algorithm become more straightforward to introduce, update and/or remove as the need arises.

The variability concerns show almost identical, if more pronounced, disparity between the AO and OO designs, again reflecting the need to explicitly call out to the delegate classes at many and various points in the core algorithm for the OO implementation, whereas the AO design can exert similar changes from outside the core implementation. The supplemental effect concerns are similarly amplified, with the AO design concentrating the optional effects in one to three operations where the OO design consistently requires five or more, again reflecting the additional overhead involved in binding these effects using OO methodology.

Concern Diffusion Over Lines of Code

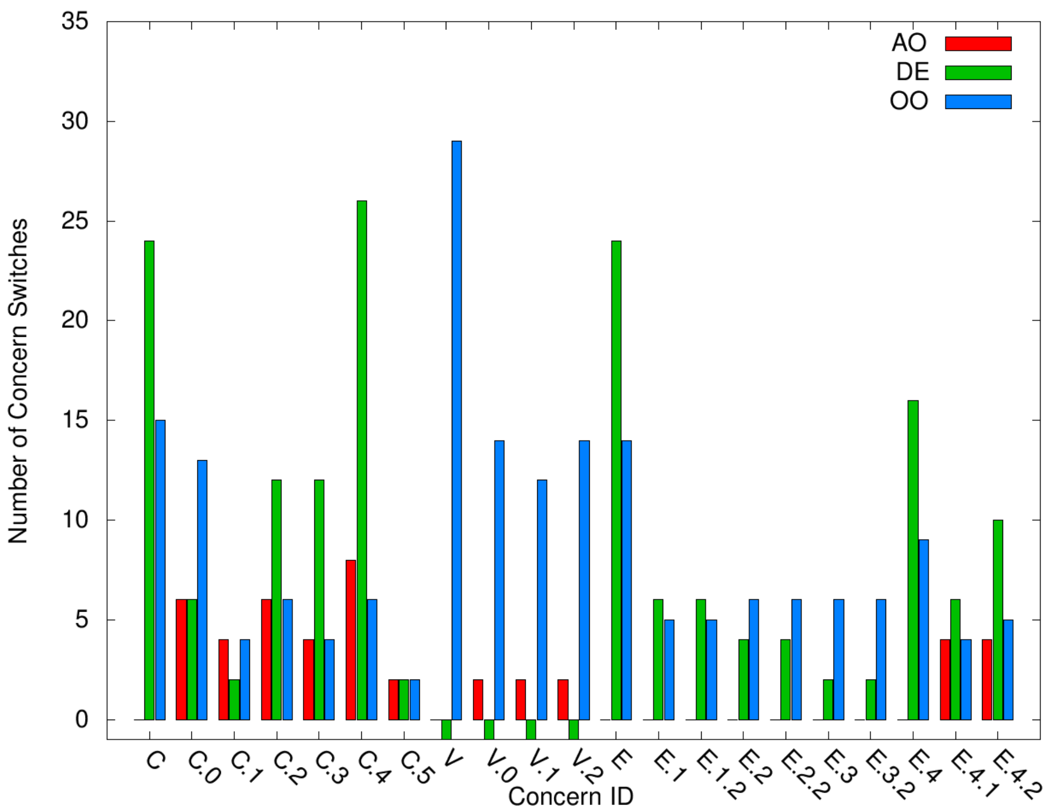


Figure 6.3: Traffic Estimator: Concern Diffusion over Lines of Code. See Table C.1 for the full listing of the illustrated concerns.

Lastly, the CDLOC scores in Figure 6.3 show the most pronounced differences between each of the three techniques. The core concerns again show the most variation, but overall both the AO and OO designs demonstrate significantly reduced source-level diffusion over the original implementation. The two exceptions to this are:

- **C.0** (Infrastructure interfaces, configuration, initialization), wherein the OO implementation shows a marked increase in diffusion over lines of code due to the need to introduce call hooks for each delegate into the configuration and initialization methods of the OO Traffic Estimator. This reflects the chief drawback[2] approach as compared the AO approach, that the core algorithm must be explicitly instrumented with calls into the delegate classes for them to be able to extend or replace the core functionality.

- **C.1** (Determine forward path of the vehicle), which records a slight increase for both the AO and OO solutions. This increase is due to the introduction of extra methods during the initial refactoring step as discussed relative to the CDO results. While curious, this is not generally a limitation of the proposed approaches; rather, it reflects suboptimal functional decomposition in the original implementation.

Otherwise, and consistent with previous discussion, the AO solution shows significantly less diffusion than its OO counterpart, with the single exception of concern **C.4** (estimate in-lane speed of lead vehicle), which concentrates the declaration and configuration of pertinent variables in an external class, relieving the `OOTrafficEstimator` class of some tangling between core sub-concerns. This is an interesting result in that it points to a possible benefit to treating some sub-parts of the core algorithm by the same means that allow the introduction of supplemental effects.

The variability concerns are once again consistent with, but more pronounced than, the other diffusion metrics, as the individual calls out to delegate classes inject lines of code in several places in the OO implementation, where the AO implementation does not, and thus remains separate. The supplemental effect concerns also generally show improvement for both the AO and OO implementations, with the AO implementation all but eliminating diffusion of individual supplemental effects, which is again consistent with earlier results.

The most significant result is indicated by the aggregate **C**, **V**, and **E**, concerns, which have flat lines where the red (AO) bars might be, indicating zero diffusion among core, variability and extended effect concerns. In other words, the AO implementation manages to completely disentangle the three broadest categories of concerns that typify the methodology proposed in this thesis. While not as profound, the OO implementation also makes considerable improvements when viewed in this way, reducing the source-level diffusion of core and supplemental effects by an average of 40%, further reinforcing the potential benefits of the *primary* vs. *supplemental* methodology.

[2] Whether explicit calls out to delegate classes is truly a "drawback" of the OO approach is ultimately a function of the skills of the development team. If the team is not comfortable with the subtleties of implicit invocation as in the AO approach, then the more explicit nature of "calling out" may provide useful guidance as to what is "going on" in the source code. Such extra calls will still "clutter" the source code, however, making it generally more difficult to understand the purpose of the *core algorithm*, which is the focus of this measurement technique.

6.3 Precedence Estimator

Concern Diffusion Over Components

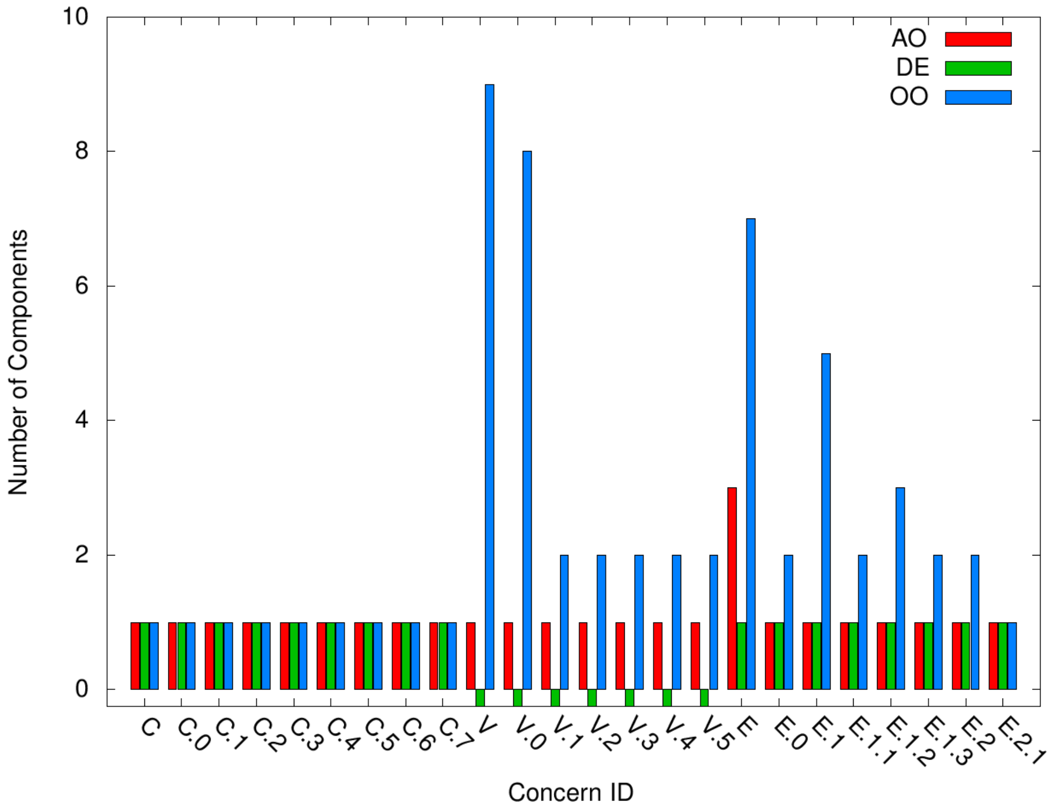


Figure 6.4: Precedence Estimator: Concern Diffusion over Components. See Table C.2 for the full listing of the illustrated concerns.

Figure 6.4 illustrates the CDC scores for the three implementations of the Precedence Estimator, which are highly consistent with the results from the Traffic Estimator. That is, there is some small but expected increase in diffusion over components for both the AO and OO designs, due to breaking up the original single-class implementation into several sub-modules.

The most important difference is that the core concerns are diffused over exactly one component in all three implementations, where the `OOTrafficEstimator` instead exhibited an increase in this type of diffusion. In this case, the supplemental effects in the Precedence Estimator did not require parts of the core algorithm to be completely masked, as was the case for the Traffic Estimator, so none of the core algorithm had to be migrated into delegate interface classes. Instead, the delegation interfaces for the Precedence Estimator were limited to simple hooks, such as in the `ObstacleUpdateProcessDelegate`, or else had trivial base cases of "return true" to be overridden by supplemental effects, such as for the obstacle candidacy tests in `ObstacleClassificationDelegate`. The simplicity of these interfaces, meant that the core algorithm remained entirely within the `OOPrecedenceEstimator`, which yields the apparent reduction in diffusion over components. This did not, however, have the same effect on diffusion over operations or lines of code, as discussed below.

Concern Diffusion Over Operations

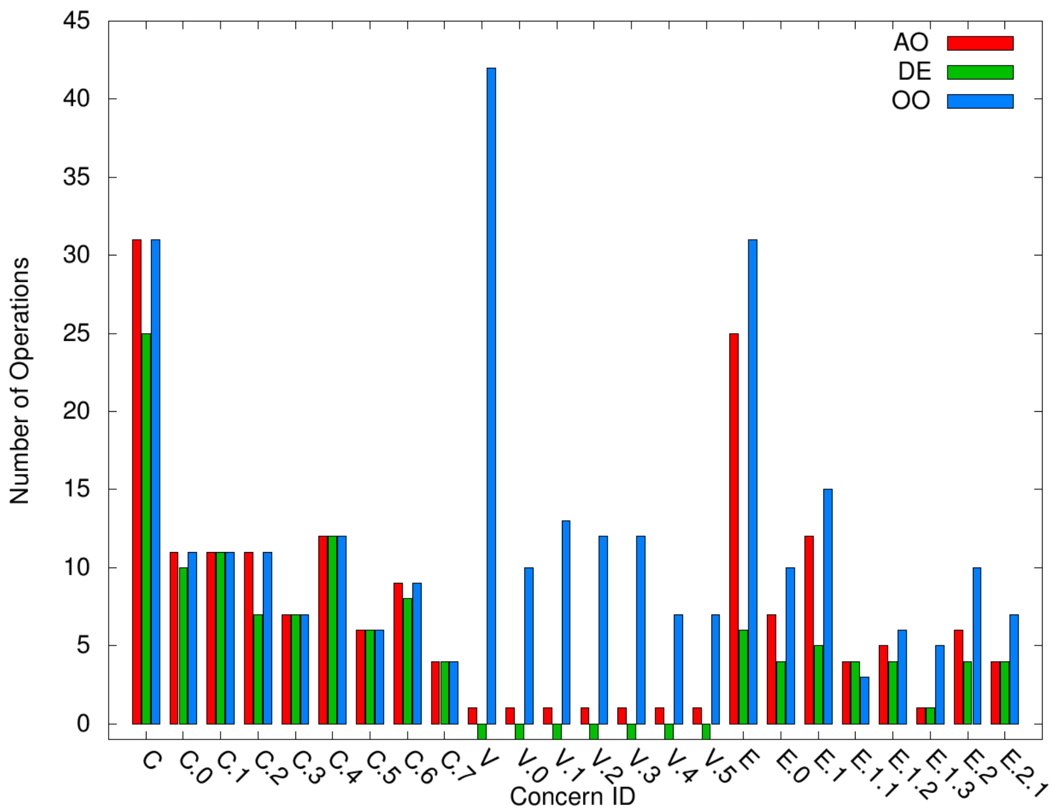


Figure 6.5: Precedence Estimator: Concern Diffusion over Operations. See Table C.2 for the full listing of the illustrated concerns.

The CDO scores for the three Precedence Estimators, shown in Figure 6.5, again show similar general trends to those of the Traffic Estimator, exhibiting small, but expected, increases in diffusion due to necessary method-level refactoring. The diffusion of the fine-grained concerns remains reasonably small, with the AO implementation showing significantly less diffusion in the majority of cases. The one notable exception is **E.1.1**, which, in the OO version, is combined in a single method of `OOMP_ObservedMovingEffects` that also implements **E.1.3**. These two concerns are instead maintained as separate advice directives in the AO implementation, yielding a slight increase in the AO diffusion over operations for the aggregated **E.1** concern.

Concern Diffusion Over Lines of Code

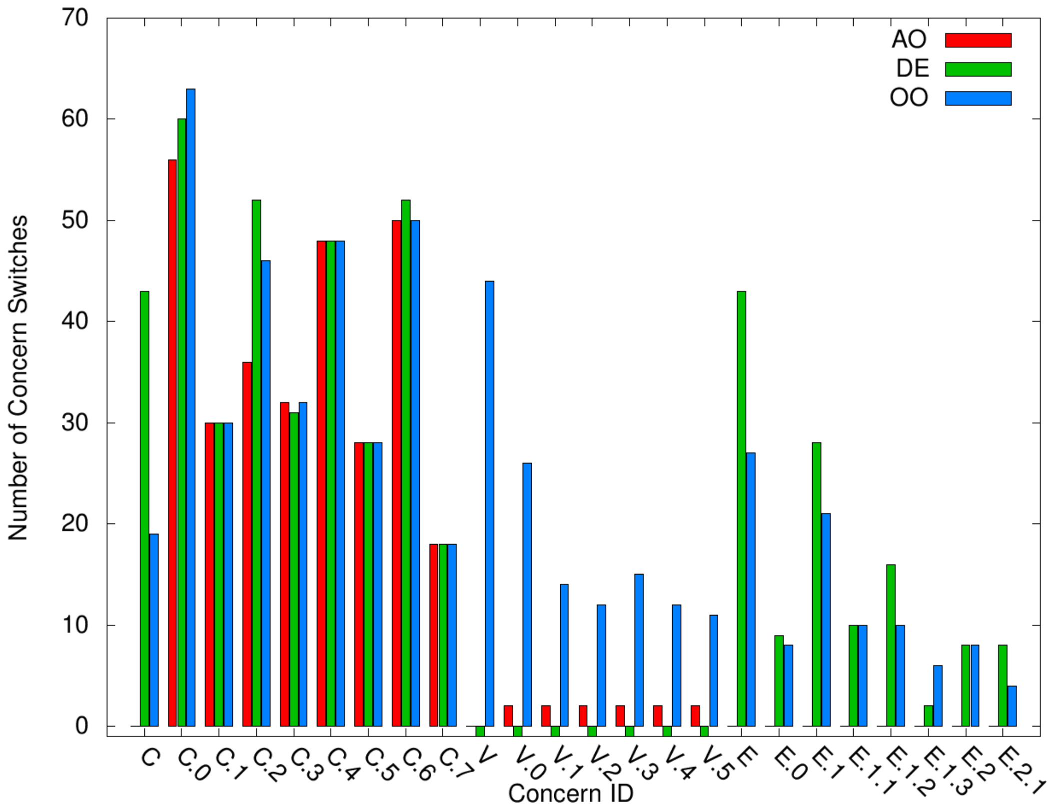


Figure 6.6: Precedence Estimator: Concern Diffusion over Lines of Code. See Table C.2 for the full listing of the illustrated concerns.

Lastly, the CDLOC scores in Figure 6.6 also follow the trends seen in the Traffic Estimator, with both the AO and OO designs generally demonstrating reduced source-level diffusion over the original implementation. The AO implementation is also generally less diffused than the OO implementation, and it demonstrates zero source-level diffusion among aggregated concerns.

The single notable exception to this norm is **E.1.3**, which shows a significant increase in source-level diffusion for the OO design. In the original implementation, this effect was implemented by two adjacent lines of code, nested deep within a complex branching logic structure. While difficult to identify and extract, this yielded a CDLOC score for the baseline implementation of only 2. In contrast, the matching score for the OO implementation included two counts associated with the generic OO overhead, plus four counts associated with sharing a method with the implementation of **E.1.1**, discussed above, yielding a total score of 6.

This can be seen as a trade-off for keeping two effects of the same datum (`isObservedMoving`) on the same point of adaptability (yield relevance test) in a single place. On one hand, the individual effects remain somewhat tangled at the source level, but, on the other hand, they are more coherent at the method level. Which is "better" is a decision to be made by the designer, and can depend on how closely related the effects are, how volatile they are expected to be, and will also include a degree of personal preference. In this case, the effects were very closely related, and the personal preference was to avoid the excessive levels of delegate specialization, so they were kept in the same method.

Otherwise, the only anomaly in Figure 6.6 is the slight increase in the diffusion of concern **C.3** for both the AO and OO implementations, which are caused by the addition of several refactored methods each core class to make it easier (or simply possible) to introduce supplemental effects. As with similar results for the Traffic Estimator, this does not indicate a particular weakness of the proposed AO or OO approaches, but rather reflects a certain sensitivity of the CDLOC metric to implementation details that are already accounted for by the CDO results.

6.4 Merge Planner

Concern Diffusion Over Components

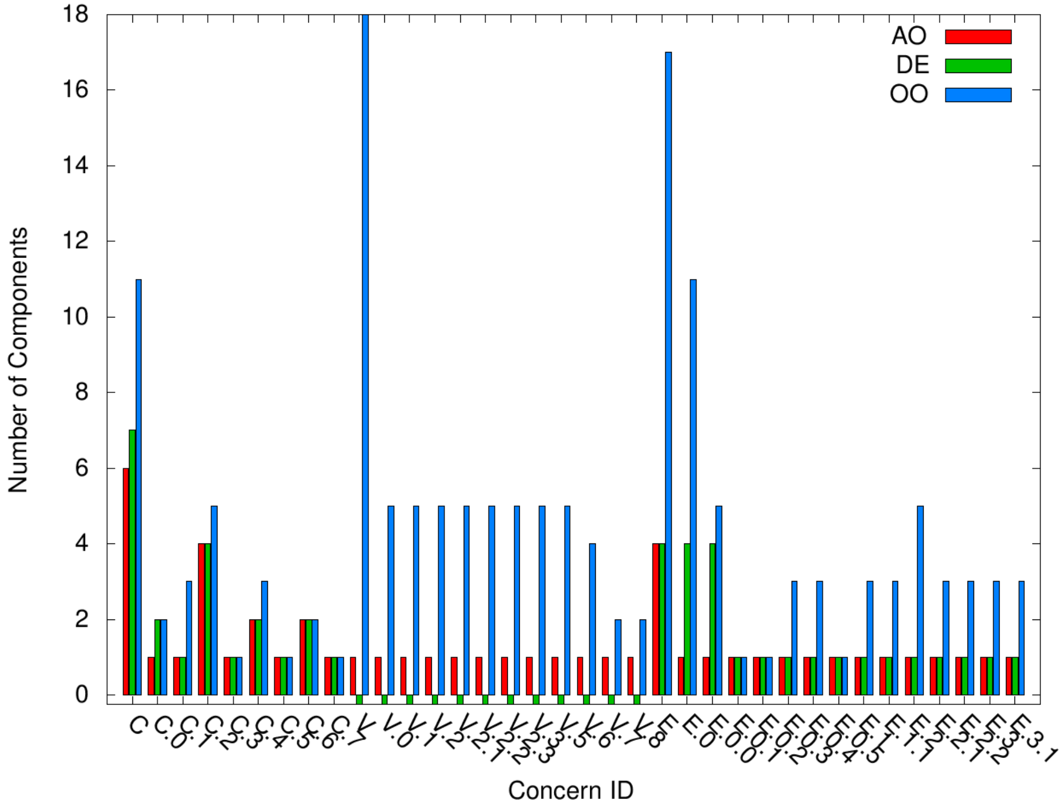


Figure 6.7: Merge Planner: Concern Diffusion over Components. See Table C.3 for the full listing of the illustrated concerns.

Figure 6.7 illustrates the CDC scores for the three implementations of the Merge Planner, with the most notable difference being the magnitude of the results, which convey a scattering of the aggregate core, variability, and supplemental effect concerns over an average of 5-6, but as many as 18, individual classes.

This is mostly due to intermediate types used in the core algorithm, which, beyond the three discussed in Chapter 5, include several others that were not affected by supplemental data. For the OO design, this creates a combinatoric explosion of extra classes that must be defined to allow and perform intermediate type extensions, which, when combined with the previously-discussed overhead of declaring delegate classes, yields the exceedingly large values for the aggregated variability and supplemental effect concerns.

Otherwise, the results are consistent with, if somewhat more pronounced than, the previous two components, with "expected" scattering of the core concern as a side effect of delegation, and the AO approach being much more consistent about keeping a 1:1 mapping from fine-grained supplemental effects to the components that implement them.

Concern Diffusion Over Operations

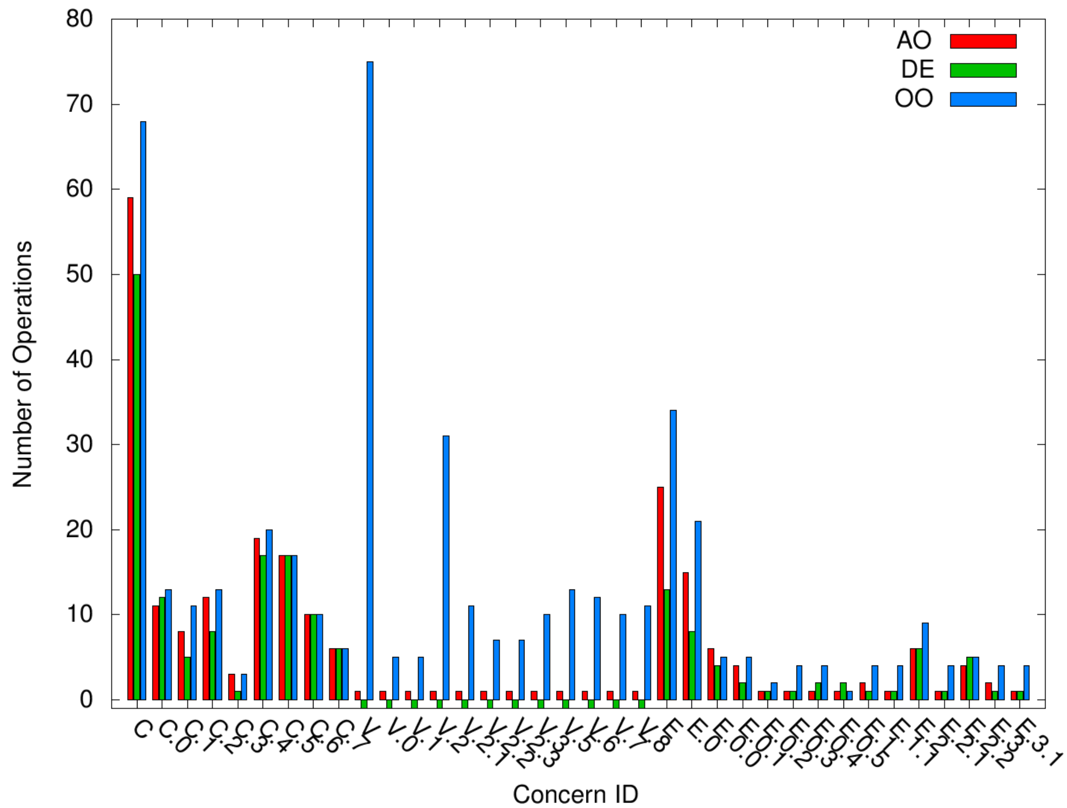


Figure 6.8: Merge Planner: Concern Diffusion over Operations. See Table C.3 for the full listing of the illustrated concerns.

The pattern of amplification in diffusion over components is repeated in CDO scores for the three Merge Planners, shown in Figure 6.8, with the results approaching diffusion over 80 operations for the many points of variation that must be exposed to allow the introduction and management of all the intermediate data types in the OO design. The analogous content for the AO design is all collected neatly into member data declarations in the XPI, which are counted as a single "operation" that is no different from declaring a collection of utility variables at the top of a method. Even if these declarations were counted as individual operations, there would be only 15 of them, one for each member of `XPI_AOMergePlanner` in Figure 5.18, which is still markedly less than 80.

Again the results are otherwise consistent with the previous components, where there is an expected increase in core diffusion over operations as a result of refactoring to create join-points, and the AO approach performs consistently better at keeping supplemental effects in comparatively few operations. The single anomaly is **E.0.0**, which shows a slight increase in diffusion for AO vs. the OO design. Upon closer inspection, this actually reflects a limitation of the AspectC++ weaver's support of the *slice* directive, which has trouble dealing with the "implicit" constructors of simple types. The workaround was to "fake" the initialization of the intermediate "isMoving" flags using advice applied to the "configuration" join-point, which scattered the implementation of **E.0.0** across 6 advice directives instead of 3. The OO equivalent was able to declare the initializers in an "inline" constructors, which limited the concern to fewer operations.

From one perspective, this might be seen as a "trivial" issue that should be ignored for the AO implementation, pending resolution of this "bug" in the AspectC++ weaver. It is, however, representative of the risks associated with adopting new technologies, so it is left in place as an important issue to consider when selecting between these techniques.

Concern Diffusion Over Lines of Code

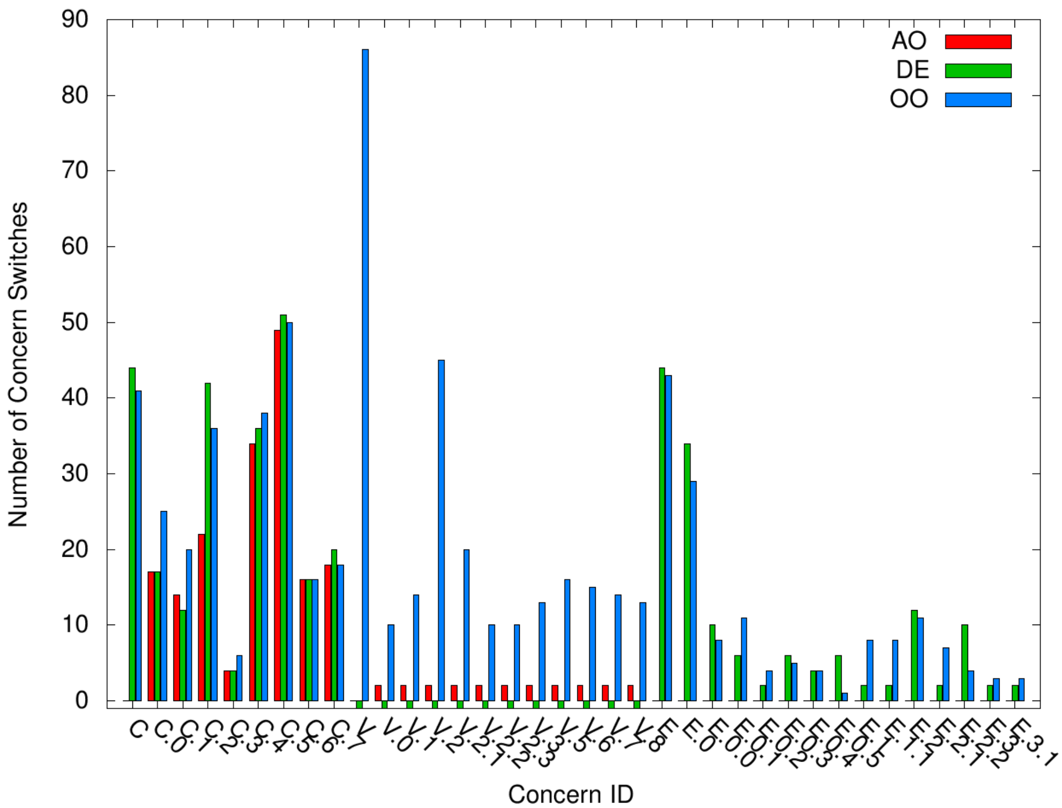


Figure 6.9: Merge Planner: Concern Diffusion over Lines of Code. See Table C.3 for the full listing of the illustrated concerns.

Lastly, the CDLOC scores in Figure 6.9 further reinforces the theme of consistent, but more pronounced, differences between the AO and OO designs, especially in the "variability" concerns, which are all co-located in the XPI for the AO design, but are instead vigorously sprinkled throughout the OO version, yielding a score of well over 80 concern switches over the whole artifact. This is a direct quantification of the critical difference between the two approaches: that "calls out" must be introduced in-line for the OO design, where the "reaching in" for the AO technique can be specified in a separate, coherent location.

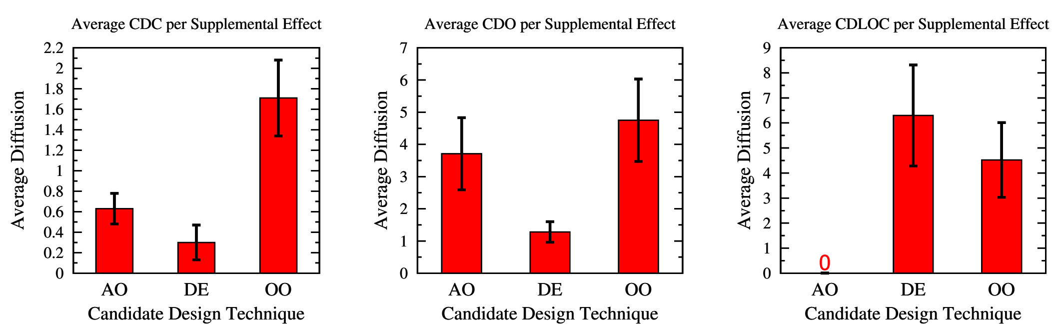
The similarity of these numbers to the CDO results is interesting in that it implies that the OO design replaced, almost in a 1:1 fashion, single-line, direct usage of various supplemental members with this "calling out" to delegate instances, in a way that the change in "conceptual load" when reading through the source is almost a zero-sum game. The "obliviousness" of the AO approach extracts this load from the core algorithm, making it "easier" to understand by itself, at the expense that the developer must "know" that some of the core algorithm may be subject to AO introduction. Whether this is truly "better", or even fundamentally "different" remains in open debate[52], but, at least in terms of being able to organize the software by the influence of individual data, these results point toward a more "natural" fit of AO techniques to the problem.

Still, as evidenced by the slight ($\sim 10\%$) reduction in diffusion of the aggregate core and supplemental effect concerns, the OO approach manages to yield some measurable benefit in terms of *tangling* concerns, which further underscores the validity of the overall *primary* vs. *supplemental* methodology.

6.5 Summary

The critical goal of the analysis in this chapter is to determine the extent to which the various concerns in each of the three behavioral modules in this case study are *diffused* over components, operations and lines of code. The relative merits of each proposed OO and AO approaches are compared to the original, direct encoding through detailed discussion of trends and exceptions within each module, highlighting both similarities and critical differences as compared to other modules.

As the central focus of this thesis is the extraction of *supplemental effects* from a set of *core algorithms*, a natural question to ask at this point might be: "How much diffusion does a *typical* supplemental effect compel in each design?" To answer this question, Figure 6.10 presents the average diffusion results for the coarse-grained "supplemental effects" concerns[3] for each of the three candidate designs.



(a) Diffusion over Components (b) Diffusion over Operations (c) Diffusion over Lines of Code

Figure 6.10: Average Diffusion per Supplemental Effect.

[3] That is, the "E" columns from each preceding figure in this chapter, normalized by the total number of supplemental effects enumerated in Chapter 5.

The results for Concern Diffusion over Components (Figure 6.10a) highlights two important points from this chapter. First, it shows that both the AO and OO technique “diffuse” the supplemental effects over more components than the original, direct encoding of the algorithms, which is an expected result of refactoring functionality from one class into severs. However, the second important observation to make is that where the AO technique allowed supplemental effects to be grouped into a small number of aspects[4], the OO technique instead required closer to two class definitions per supplemental effect. This is primarily caused by the difficulty, especially in the Merge Planner, of introducing member data into existing data classes using strictly OO techniques, where the AO inter-type declaration mechanisms were a much more natural fit.

To a certain extent, this pattern is reflected in Figure 6.10b, but in this case both versions show more significant increases in diffusion over operations. This was primarily due to refactoring otherwise simple logic statements in the core algorithm into dedicated method calls to allow interception by AO advice directives or redirection to OO delegates. While the AO design still showed less diffusion than the OO design, the key insight to be drawn from this is that, regardless of the specific technique for applying supplemental effects, the overall methodology requires significant design consideration and incurs nontrivial overhead in the detailed implementation.

The most compelling results are presented in Figure 6.10c, which shows that, at least at the coarsest granularity, the AO design completely eliminated diffusion of the supplemental effects in all three three behavioral modules in this case study. In contrast, the OO design only slightly decreases diffusion on average, and its large standard deviation indicates somewhat erratic results, with some cases that improve diffusion, and others that may not. Still, this is promising in that, even within the limitations of established languages and techniques, the proposed *primary* vs. *supplemental* methodology can yield software that is more coherently arranged around the influence of individual data. In turn, this limits the amount of software that must be understood in order to correctly perform the types of adaptation laid out in Chapter 2.

This enhancement to source-level understandability is certainly a benefit of the methodology proposed in this thesis, and, all else held equal, the AO and OO designs would be clearly preferable to the original implementation. There are, however, other facets of this problem, such as whether the additional diffusion over components and operations imposes a larger cost to the development process than can be offset by the benefits of encapsulating the supplemental effects in separate modules. In pursuit of an answer to this question of relative costs and benefits, the next chapter presents an analysis technique that binds issues of software design to economic theory in order to derive a “value” for each design in terms of how well it can accommodate future adaptation.

[4]0.6 components per effect means an average of 1.7 effects per component.

Chapter 7

Results: Net Option Value

This chapter complements the source-level "concern diffusion" results, which are promising in terms of separation of core and supplemental concerns, with a more abstract analysis of how well each design would accommodate adding, removing or substituting individual functional modules, especially those pertaining to supplemental effects. Section 7.1 begins with a thorough introduction to this "Net Option Value" analysis technique, which is based on a highly parameterized model of the economic "value" provided by a given design. The estimation of the parameters of this model is a critical and somewhat subjective process, so Section 7.2 reviews precedents in the literature for deriving these parameters from measurable properties of a software artifact. Variations on these parameter estimation techniques are explored in Section 7.3 before arriving at the final results for this chapter, which are summarized in Section 7.4.

7.1 Introduction: The Net Option Value of a Modular System

Net Option Value (NOV) is an analytical model introduced by Baldwin and Clark[5] that attempts to measure the relative merits, or "values", of alternate designs for a given system. The value that a design provides is modeled in terms of so-called "real options", which is related to economic theories wherein an informed actor would invest resources "now" in order to have the opportunity to evaluate and select the "best" among an unknown set of options that will present themselves "in the future". Baldwin and Clark tie this idea to technological systems by recognizing that the act of subdividing a system into separate modules (an investment "now") creates a set of options for experimenting with the implementation of each individual module, the "best" of which may be selected and integrated into the final system "in the future".

Clearly, some designs will support this process better than others, and Baldwin and Clark's model attempts to account for the salient differences that contribute to alternate designs being better or worse than one another. They propose that the overall value of a design is the summation of the value provided by each individual module in that design, and that the value of an individual module can be modeled in terms of:

- The costs incurred to develop and test new candidate implementations of the module;

- The expected benefits that may be gained by selecting the best such candidate;
- The costs incurred to integrate the new version into the rest of the system.

The nuances of their formulations are critical to understanding the model, so they are presented here in some detail. The benefits model is presented first, followed by the cost models, before presenting the overall NOV model at the end of this section.

Benefits

To model the expected benefits, Baldwin and Clark assume that the benefit expected from from a single (re)design experiment on a single module can be modeled by a normal distribution whose mean is the value of the "current" implementation, which is normalized to zero, and whose variance is determined by two intrinsic properties of the module:

1. The *complexity*, η, of the module, which supposes that modules that fill larger, more complex roles will generally have more "room for improvement", and
2. The *technical potential*, σ, of the module, which is tied to the idea that, independent of complexity, two modules may contribute different amounts of end-user value within a given design, such as user interface modules vs. hidden utility modules.

The Canonical Module

The benefit of generating multiple candidate replacements is modeled as the expected maximum value of the corresponding number of draws from the aforementioned normal distribution. Positive draws are treated normally, and negative draws are assigned a value of zero, as there is always the option of keeping the "original" module if an experimental version turns out worse. This is the most confusing aspect of the formulation, which is often misinterpreted to mean that only the positive half of the distribution is considered. This is mathematically distinct from considering the whole distribution, but assigning the left half a value of zero for the purposes of computing an expected value. To address this confusion, the derivation of this function, which is denoted $Q(k)$, is presented in Equation 7.1. A closed-form solution to the integral(s) in this equation is not feasible, and a lookup table, derived by numerical integration and reproduced in Table 7.1, is used instead.

$$\begin{aligned}
\text{let } N(x) &= \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad \text{(standard normal dist.)} \\
\text{let } Z(i) &= p(N(x) < i) = \int_{-\infty}^{i} N(x)\mathrm{d}x \\
\text{def } Q(k) &= E\{\max(0, k \text{ draws from } N(x))\} \\
&= \int_{-\infty}^{\infty} \max\{0, x\} N(x) p(\text{k-1 other samples} < x)\mathrm{d}x \\
&= \int_{0}^{\infty} x N(x) Z(x)^{k-1} \binom{k}{k-1} \mathrm{d}x \\
&= \int_{0}^{\infty} x N(x) Z(x)^{k-1} k \mathrm{d}x
\end{aligned} \tag{7.1}$$

k	0	1	2	3	4	5	6	7	8	9	10
$Q(k)$	0.00	0.40	0.68	0.89	1.05	1.17	1.27	1.35	1.42	1.49	1.54

Table 7.1: $Q(k)$: The expected maximum of k draws from a normal distribution, assuming negative draws are assigned a value of zero.

One possible point of contention about this representation is that it assumes a pre-existing module whose "value" can be treated as the mean or nominal value to be had from any other version of that module. The rationale is that the realization of the system requires at least one implementation of each module, so the costs associated with the generation and integration of that initial version can be treated as fixed or otherwise sunk costs relative to future adaptation. Yet, it is often the case, especially in more industrial settings, that a firm would consider soliciting multiple versions of a given module without already having this "baseline implementation" in hand. This slightly alters the derivation of $Q(k)$, and it is worthwhile to investigate how much the assumed existence of a baseline implementation affects the benefits model. Holding the other modeling assumptions fixed, namely that:

1. There will be k independent design experiments,
2. Whose ultimate values will be still normally distributed,
3. Around *some* nominal value that can still be normalized to zero,

The new formulation of $Q(k)$, denoted here as $R(k)$, changes in subtle but important ways:

$$\begin{aligned}\text{def } R(k) &= E\{\max(k \text{ draws from } N(x))\} \\ &= \int_{-\infty}^{\infty} xN(x)p(\text{k-1 other samples} < x)\mathrm{d}x \\ &= \int_{-\infty}^{\infty} xN(x)Z(x)^{k-1}k\mathrm{d}x \qquad (7.2)\end{aligned}$$

The primary difference between $R(k)$ and $Q(k)$ is the extension of the lower bound of the integral from zero to $-\infty$. Intuitively, it is expected that $R(k)$ will have lower initial values, but will eventually "catch up" to $Q(k)$, as the ultimate result of the *max* operator will be the same after the first non-negative sample. To illustrate this, Figure 7.1 plots both $Q(k)$ and $R(k)$ for $k \in [0, 20]$.

As shown in Figure 7.1, $R(k)$ does, in fact, catch up very quickly to $Q(k)$, reaching a very small margin at $k = 4$, and being largely indistinguishable for $k \geq 5$. This is reassuring in that the assumption of a baseline implementation does not bias the result as much as it provides a "head start", and that initial lead is consumed in comparatively few samples. Moreover, most modules' NOV curves typically peak at $k \geq 3$, further diminishing the potential difference between assuming a baseline of zero and no baseline at all.

From the perspective of the work at hand, there is expected to be a pre-existing implementation of a given component that must, for one reason or another, be adapted to accommodate certain platform-specific details. This adds credence to presumption of a baseline implementation for these experiments, but the preceding discussion shows that the formulation does not lose generality if these issues are to be considered at design time, and

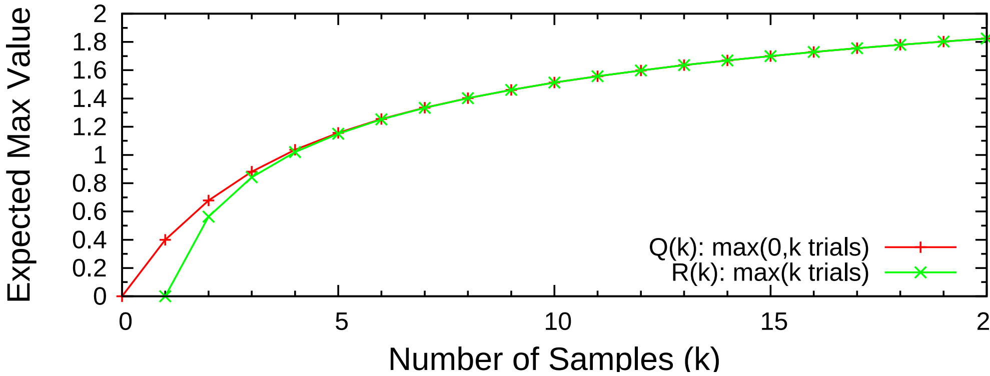


Figure 7.1: Expected maximum value of k draws from a normal distribution, assuming both a minimum value of 0 ($Q(k)$) and no minimum value ($R(k)$)

they may even be used to help inform the allocation of research and development funding for a design that is "still on the drawing board".

Not All Modules Created Equal

As with many other analyses based on the normal distribution, the standard deviation can be applied after the fact, by simply scaling the outcome of the computation. That is:

$$Q(k)|(\sigma = s) = s(Q(k)|(\sigma = 1)) \tag{7.3}$$

In the NOV model, the standard deviation represents how much a given module might be improved to yield higher value, where higher standard deviations have higher such potential. The complexity of an individual module, η_i, makes a square-root contribution to the standard deviation of that module, and, as the total system complexity is typically normalized to one, this nicely encodes the idea that, all else held equal[1], "more modules is generally better". Mathematically, consider the example of a two-module system with nonzero complexities A, B that add to 1:

$$\begin{aligned} & A + B = 1, \{A, B\} > 0 \\ \rightarrow\ & 0 < \{A, B\} < 1 \\ \rightarrow\ & \sqrt{A} > A, \sqrt{B} > B \\ \rightarrow\ & \sqrt{A} + \sqrt{B} > A + B \\ \rightarrow\ & \sqrt{A} + \sqrt{B} > 1 \end{aligned} \tag{7.4}$$

This is summarized in economic terms as "a portfolio of options is generally better than an option on a portfolio",[5] which is highly intuitive, economically sound[2], and thus universally accepted as part of the model. The complexity term is then multiplied by the

[1] "All else equal" implies that there are few inter-modular dependencies generated by the subdivision, and all sub-modules have an equal potential to contribute to end-user value.

[2] The model even captures the idea of "diminishing returns", where the costs associated with maintaining a large portfolio of small options outweighs the benefits of a smaller portfolio of larger options. This is accomplished by subtracting a linear cost function (see Equation 7.6) from the cost curve in Figure 7.1.

technical potential, σ_i, of the module, and the overall benefit of k experiments on a given module i is given as:

$$B_i(k) = \sigma_i \eta_i^{\frac{1}{2}} Q(k) \tag{7.5}$$

The difficulty of selecting and justifying the parameters in this equation is recognized as the single greatest challenge for applying the NOV model to a set of candidate designs[5, 34, 55]. Although concrete advice for doing so remains somewhat sparse, there are several simplifying assumptions and other historical precedents that are commonly used to simplify the problem, as reviewed in Section 7.2.

Costs

Along with the benefits, the NOV model also accounts for the various development and testing costs associated with modular experimentation. The vast majority of such costs are tied in [5] to the complexity of a module, where more complexity generally incurs more cost. In the general case, the per-experiment cost for a given module i is modeled as a module-specific function of complexity, denoted $C_i(\eta_i)$. The cost model ignores any learning that takes place across multiple experiments, so the cost to perform k experiments is simply:

$$C_i(\eta_i)k \tag{7.6}$$

The final element of the NOV calculation for a given module is called the "visibility cost", and it accounts for the effort of integrating that module into the surrounding system. This cost is determined by a system's Design Structure Matrix, which denotes dependencies as X's in a grid representation of the "design parameters"[3] (DP's) of a system, as shown in the example DSM in Figure 7.2.

	DE Traffic Estimator		1	2	3	4	5	6
Core Alg.	Estimate Lane Speed	1						
	Select Closest Bumper	2	X					
	Identify Vehicles On Forward Path	3		X				
	Determine Forward Path	4			X			
	Configuration and Initialization	5						
	Class Definition	6	X	X	X	X	X	
Env. Params	Pose	7	X	X	X		X	X
	Velocity	8	X					
	Size	9		X	X			
	Is Moving	10	X					
	Is Observed Moving	11	X					
	Is Predicted	12	X					
	Lane Associations	13	X		X		X	X

Figure 7.2: DSM for original Traffic Estimator, listing various "design parameters" on the left side, and showing notation for dependencies(X's) and modular clusters(bold boxes).

[3]In the most abstract sense, a "design parameter" is something that must be addressed in order to realize a system, which is closely related to the concept of a "concern" from Chapter 6. To avoid confusion with the input parameters to the NOV model, "design parameters" will be referred to as "DP's" hereafter.

For each module j that depends on, or "sees" module i, which implies an "X" in row i, column j of the DSM, an integration cost proportional to the complexity of the dependant module j is incurred. The coefficient is simply denoted c_j, yielding the mathematical representation of visibility cost:

$$Z_i = \sum_{j \text{ sees } i} c_j \eta_j \tag{7.7}$$

It is worth noting that the visibility cost described by Equation 7.7 is *not* a function of the number of experiments (k), as were the previous two components of the NOV calculation. Instead, it is a fixed function of the inter-module dependencies described in the system DSM. This captures three important aspects of typical system development:

1. Many (k) individual design experiments may occur within an individual module before it is integrated into the rest of the system (once);

2. The principal costs associated with integrating the "new" module into the rest of the system are in updating and (re)validating its dependant modules;

3. These costs may be large enough to overcome the "benefit" derived from improving the module in the first place, such as for "highly visible" modules.

Putting it All Together

The benefit and cost models are combined into a single model for the net value of k_i experiments on a given module as:

$$V_i(k_i) = \sigma_i \eta_i^{\frac{1}{2}} Q(k_i) - C_i(\eta_i)k_i - Z_i \tag{7.8}$$

The Net Option Value for that module is then the maximization of $V_i(k_i)$, or:

$$NOV_i = \max_{k_i} \left\{ \sigma_i \eta_i^{\frac{1}{2}} Q(k_i) - C_i(\eta_i)k_i - Z_i \right\} \tag{7.9}$$

Finally, the Net Option Value for the whole system is taken to be the sum over all modules:

$$NOV = \sum_i \left\{ \max_{k_i} \left(\sigma_i \eta_i^{\frac{1}{2}} Q(k_i) - C_i(\eta_i)k_i - Z_i \right) \right\} \tag{7.10}$$

This is the most generic form of the NOV calculation, and it is meant to accommodate arbitrary benefit and cost models that can be tailored to suit any system. The major drawback to this generic form is that it requires the specification of three scalar parameters (σ_i, η_i, and c_i) and one cost function ($C_i(\eta_i)$) for each module in each design of a given system. This is widely recognized as the greatest challenge in applying NOV to novel systems[5, 34, 55], and there are several common approaches, presented in the following section, that are typically employed to simplify the problem.

7.2 Precedents for Parameter Selection

Complexity

The first and most common simplification of the NOV parameter estimation problem is to allocate complexity according to the number of DP's that comprise each module. For a design that consists of N such DP's, a complexity of $\eta = \frac{1}{N}$ is assigned to each DP, which ensures that the overall system complexity is normalized to one as discussed above. Modules are defined as a group of one or more of these DP's, and for a module i that embodies M_i DP's, complexity is simply:

$$\eta_i = \frac{M_i}{N} \tag{7.11}$$

This is often criticized as an overly-simplistic assumption, as no two DP's are likely to impose identical amounts of complexity in a "real" system. When analyzing existing systems, results from analysis metrics such as source code size might be used to distribute complexity more effectively, but such metrics are not available at design time for new systems, obviating their usefulness in the general case. Moreover, the sensitivity of the NOV calculation to variations in complexity distribution is not well understood, so the simplified derivation of complexity in Equation 7.11 holds as the de facto standard.

For the behavioral components under investigation, it is not clear that allocating the complexity purely by DP count will fully capture the differences between modules that implement core algorithms and supplemental effects, so the sensitivity experiments presented in Section 7.3 include a comparison of NOV results using this simple computation of complexity to results attained by allocating complexity according to the number of lines of code per-module, leveraging the hand-labeled source code used to compute the "concern diffusion" metrics discussed Chapter 6.

Costs

As with the complexity model, Baldwin and Clark make some simplifying assumptions regarding experimental and visibility costs in their example analyses. First, they concede that the cost of an experiment on a given module can theoretically be an arbitrary and unique function of that module's complexity, but they assume for their analysis that cost is simply proportional to complexity, or:

$$C_i(\eta_i) = c_i\eta_i \tag{7.12}$$

Moreover, they assume that the ratio of cost to complexity, c_i, is simply 1, arguing that variations in proportionality can be represented by alternate allocations of complexity. This further simplifies the cost function to the identity function:

$$C_i(\eta_i) = \eta_i \tag{7.13}$$

These, combined with the "break even" assumption, discussed below, are once again highly simplistic, but Baldwin and Clark point out that these are still effective modeling assumptions and that more complicated cost functions will not only require additional justification, but they are also unlikely to impact the *relative* values of each module. That is, NOV is already a relative valuation metric, so more complicated estimations of experimental costs are not likely necessary. This rationale is largely accepted by the design community and this simplified form of experimental cost is used in many other NOV analyses.

The scale factor from Equation 7.7, c_j, is also typically omitted for similar reasons, simplifying the visibility cost for module i to:

$$Z_i = \sum_{j \text{ sees } i} \eta_j \tag{7.14}$$

This is once again simplistic, but it also goes uncontested in the design community, as it is consistent with preceding assumptions and eliminates an additional set of parameters that must be derived and justified per-module. Collecting these assumptions into Equation 7.10 yields the simplified version of the NOV calculation that will be used to generate all NOV results hereafter:

$$NOV = \sum_i \Big\{ \max_{k_i} \big(\sigma_i \eta_i^{\frac{1}{2}} Q(k_i) - k_i \eta_i - \sum_{j \text{ sees } i} \eta_j \big) \Big\} \tag{7.15}$$

This simplified NOV formulation is now a function of only:

1. Module complexity, η_i, whose typical derivations are described above,
2. The expected value of k experiments, $Q(k)$,
3. The inter-module dependencies described in the DSM, and lastly
4. Technical potential, σ_i, whose derivation is discussed below.

Technical Potential

The final simplification proposed by Baldwin and Clark is the so-called "break-even" assumption, which provides a baseline against which all candidate designs may be judged by assuming that:

- A *single* experiment ($k = 1$)
- on an *unmodularized* system ($\eta_1 = 1, Z_1 = 0$)
- breaks *even* ($NOV = 0$)

When applied to Equation 7.15, these assumptions yield a specific value for the technical potential of this theoretical monolithic design, which is used as the maximum technical potential for any module in any alternate design.

$$\begin{aligned} \sigma_1 \eta_1^{\frac{1}{2}} Q(1) - 1\eta_1 - Z_1 &= \quad 0 \\ \Rightarrow \sigma_1 (1^{\frac{1}{2}})(0.4) - 1 - 0 &= \quad 0 \\ \Rightarrow \sigma_1 &= \quad 2.5 \end{aligned} \tag{7.16}$$

It is tempting, but inaccurate, to think that specifying $\sigma_1 = 2.5$ means that the unmodularized design has the potential to be 2.5 times "better" than it "currently" is. The value of 2.5 merely scales the expected benefits model from Equation 7.5 so that the theoretical base case of implementing the entire system as one giant module yields a mathematically-convenient Net Option Value of zero. This way, any other design that yields zero NOV can be thought of as being "no better than" simply clumping everything into one module, and positive results indicate an improvement over this imaginary baseline. Moreover, this rationale provides a certain reconciliation of the "units of value" in the benefit and cost models, whose relationship to each other is otherwise unclear. The use of 2.5 as the maximum technical potential for any module in an alternate design is thus widely accepted by the design community.

Opinions diverge, however, as to how to determine where the technical potential of individual modules lies between 0 and 2.5. Baldwin and Clark's examples are based on historical data from IBM, so they represent a mixture of omniscient hindsight with a certain amount of rational judgement, such as pointing out that a better-performing CPU has "much greater" technical potential than an improved case material for a computer system. While compelling in their own right, these do not provide a great deal of guidance as to how to allocate technical potential for other systems.

One interesting solution comes from work on extending the standard DSM model to account for external factors that are not under the control of the designer. These so-called Environment Parameters[55] (EP's) allow the specification and analysis of dependencies on higher-level design decisions or end-user requirements. One example from their work points out that the implementation of an alphabetization algorithm depends on the language and corresponding character encoding used in the system, but that these parameters are typically beyond the control of the designer.

The key insight is that many changes to a software system are driven by changes in such external requirements, so the value provided by a design, at least in terms of adaptation and maintenance, might be linked to how well it accommodates changes to its environment parameters. These modify the NOV formulation by scaling the technical potential of each module according to the fraction of EP's that affect it:

$$\sigma_i = 2.5 \frac{\text{No. EP's that affect module i}}{\text{Total number of EP's}} \tag{7.17}$$

Even with this model in-hand, some amount of judgement was applied to further reduce or eliminate the technical potential for modules that "obviously" had less or no potential, which still leaves room for fuzzy subjectivity in the selection of this parameter. Relative to the behavioral components under investigation, algorithmic dependencies on platform-specific data could easily be modeled along these lines, as the data that a robot provides is not generally under the direct control of component designers, and changes to that data are expected to compel adaptation, as discussed in Chapter 2. In Figure 7.2 above, there are seven such EP's, listed at the bottom of the DSM and collected under the abbreviated heading "Env. Parameters". The X's to the right of these EP's capture how the individual data in the Moving Obstacle representation influence the various stages of the original Traffic Estimator's algorithm, allowing the application of Equation 7.17 to estimate the technical potential of each module in the design.

In more recent work on NOV modeling of aspect-oriented designs[35, 34], several variations on this estimate of technical potential were attempted in order to accommodate more finely-grained design models. In particular, these models had many more EP's than in previous work, along with relatively sparse dependencies that led to artificially low values for technical potential. They proposed an alternate form of Equation 7.17 that accounts for these issues by introduce an explicit bias and scale factor for "end-user visibility", p_i:

$$\sigma_i = (1 + \frac{\text{No. EP's that affect module i}}{\text{Total No. of EP's}}) \times p_i \tag{7.18}$$

This equation was scaled to have a maximum value of 2.5, possibly via ensuring $p_i \in [0, 1.25]$, to yield the first set of values they tested. Ultimately, however, they chose to use a technical potential of 2.5 for most modules, setting the rest to zero based on their own judgement, demonstrating that this maxed-out estimation of technical potential yielded NOV results that had higher contrast while retaining consistent ordering between alternate designs as compared with other, more complicated techniques.

All of these studies point out that additional sensitivity analysis is necessary, and that alternate parameter estimation techniques may yield substantially different and/or more compelling results. As such, several variations on the above estimation techniques are evaluated in the following section in order to understand how they affect the NOV results, including how well each technique differentiates the strengths and limitations of each design.

7.3 Experiments in Parameter Estimation

The primary inputs to the parameter estimation experiments are the DSM's for each implementation of the urban driving components discussed in Chapter 5. These DSM's, which are presented in Appendix D for reference, were determined in advance of these parameter experiments and remained fixed throughout the process. For the original components, the DP's represent significant stages of the core algorithm, and are analogous to the fine-grained core "concerns" listed in Appendix C. The AO and OO versions extend these DP's to account for the exposure of points of variation and the separate encoding of individual supplemental effects. The Moving Obstacle representation is broken out into 7 distinct EP's, one for each of the data listed in Figure 5.2, allowing the fine-grained dependencies on those data to be represented and analyzed.

The Straightforward Approach

The assumptions from [55] were used as a starting point, with each DP treated as a separate module. That is, complexity was evenly distributed across DP's as in Equation 7.11, experimental and visibility costs were assigned according to Equation 7.15, and technical potential was determined entirely according to environment visibility as in Equation 7.17. Figure 7.3 shows the NOV results under these assumptions, highlighting both the positive-only summation, which is the "true" NOV calculation, and the summation of all values, positive or negative, which is useful for understanding the overall effects of the different parameter estimation techniques.

The first thing to be noticed in Figure 7.3 is that the original, directly encoded versions of each component have significantly higher, or at least significantly less negative, value than the corresponding AO and OO implementations, regardless of whether all modules (green bars) or only positive modules (red bars) are counted. While it is certainly possible

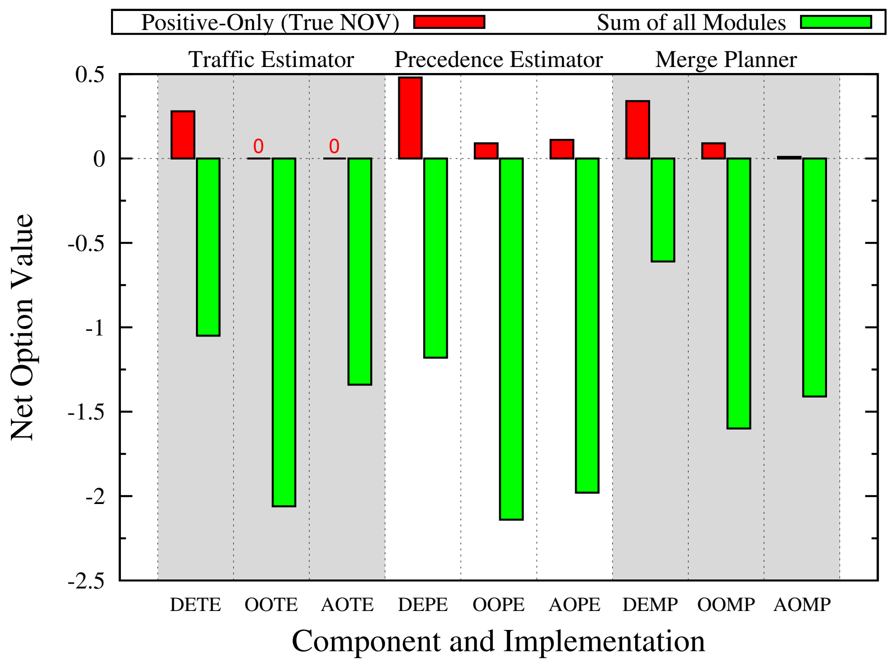


Figure 7.3: NOV results using assumptions and parameter estimation from [55]. The colored zeros along the X-axis help discriminate between designs that have zero NOV, such as OOTE and AOTE, and those that have very small, but still nonzero results, such AOMP, whose nearly-invisible bar represents a result of 0.01.

that these designs are, in fact, worse than the originals, this phenomenon is highly counterintuitive, as the dependency structures for these designs does not seem to warrant such poor performance.

Upon closer inspection, no AO or OO design has more than two modules that yield positive results, and in most cases, there is not even an initial positive slope to the NOV curve, as the benefits are completely overwhelmed by experimental costs. There are several possible causes for this phenomenon, all of which are tied either to underestimation of technical potential, or misallocation of complexity amongst the modules in the system.

Treating the DP's as individual modules, while valid from a certain perspective, may yield artificially small complexity values, possibly depressing the benefits model for some modules and boosting the experimental costs for others. This might be mitigated by a combination of aggregating the DP's along class or aspect boundaries, and/or allocating complexity according to a more concrete metric, such as source code size, both of which are explored below.

It is also worth noting that there are seven EP's in this study, where there were only three in the original work[55]. The environment dependencies in this study are also significantly less dense, which leads to diminished technical potentials, on the order of $0.4 - 0.8$, for the majority of modules, instead of $1.5 - 2.5$ from previous work. Such small values of technical potential are likely the primary cause of the strongly negative overall results (green bars) in Figure 7.3. This effect may be mitigated by using an alternate model of technical potential, such as a variation of Equation 7.18, or weighting the EP's by volatility, as was suggested but not pursued in [55].

To a certain extent, previous work has explored coarse differences between some parameter estimation techniques[34, 35], but fine-grained sensitivity analysis seems to be absent from the literature. To help fill this void, and to aid in justifying the results for this particular study, the following section explores each of the variations discussed above and analyzes their impact on the NOV results before arriving at a final parameter estimation policy for this work.

Alternate Parameter Estimation Techniques

The four proposed variations on parameter selection yield sixteen possible combinations to examine in order to fully understand their combined effects on the NOV results. Rather than explore this space entirely, each variation will be applied incrementally, and the sensitivity of the NOV results at each stage will be taken as a fair proxy of the individual effects on the NOV formulation. Using the NOV results in Figure 7.3 as a starting point, alternate parameter estimation techniques will be applied in the the following order:

1. Aggregating DP's into modules at class and aspect boundaries,

2. Introducing a minimum technical potential that is complemented by EP dependencies,

3. Weighting EP's according to expected volatility, and

4. Allocating complexity by source code size instead of by DP count.

Justifications for each variation are discussed along with the ramifications of its application, both in terms of the impact on the NOV results, and in terms of the specific design issues that are highlighted in the process. The policy for applying each variation will be selected before proceeding to the next, and that policy will hold through the final results presented in Section 7.4.

Aggregation

As mentioned above, one of the key differences between this and earlier work is the relatively fine granularity of DP's and their dependencies, which, with each DP treated as a separate module, could adversely affect the estimation of experimental and visibility costs. Aggregating the DP's along class or aspect boundaries, denoted by bold boxes in the DSM's in Appendix D, may help address this problem, the results of which are illustrated in Figure 7.4.

The results from Figure 7.3 are represented by the same colors (red and green) in Figure 7.4, and the blue and violet bars indicate the corresponding effects of aggregation, which differ from the previous results in three significant ways. First, the NOV results for the DE implementations under aggregation are all zero, as illustrated by the lack of blue and violet bars in the DE columns. This is an expected result of the "break-even" assumption discussed above, as the DE implementations are all concentrated in single classes. While this may not give fair treatment to the original versions, it will be accepted for now and revisited at the end of this section.

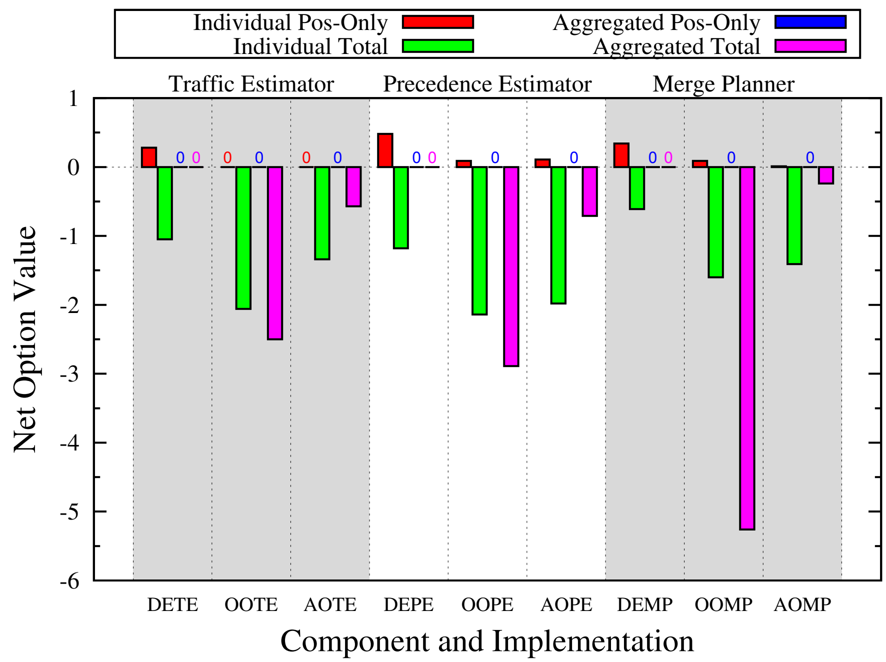


Figure 7.4: Effects of Aggregation on NOV Calculations

Second, for the absolute totals (violet bars), the ordering between the two proposed techniques is preserved (i.e. AO > OO), but the contrast between them is significantly increased. This is largely due to the clumping of the core algorithms into larger modules with correspondingly higher complexity. For the OO versions, this imposes higher visibility costs on the delegate interface classes, where the AO versions are not subjected to the same penalty. This is consistent with the "obliviousness" property of aspects, discussed in Chapter 3, which supports the aggregation of DP's into larger modules for this study.

Lastly, the complete lack of positive results for the aggregated variants (i.e. no violet bars at all) indicates that the act of aggregation eliminated the few positive values that contributed to the AO and OO implementations in Figure 7.3. As discussed above, these results were already counterintuitive and are more likely due to an underestimation of technical potential than a misallocation of complexity, so aggregation will be retained for further analysis.

Base Technical Potential

As discussed above, Equation 7.17 likely underestimates technical potential for individual modules as a function of the sparse EP dependencies in this study. This was the experience in [34] that compelled the introduction of Equation 7.18, which essentially adds 1.0 to the technical potential for all modules. While this may seem arbitrary at first glance, it is consistent with the idea that the EP's included in a given DSM may not be a complete picture of the external factors that influence a software system. For example, the EP's in this case study are limited to the current Moving Obstacle representation, but many other external factors, such as variations in the competition rules or deployment environment, would also affect these software components. It is also possible that EP's alone, even if enumerated in their entirety, cannot account for all sources of value for a module, and the extent of their impact is too extreme as described by Equation 7.17. To explore this possibility, a baseline technical potential, σ_0 will be assigned to all modules, and the complement (up to 2.5) will be filled in by EP dependencies according to:

$$\sigma_i = \sigma_0 + (2.5 - \sigma_0)\left(\frac{\text{No. EP's that affect module i}}{\text{Total No. of EP's}}\right) \tag{7.19}$$

Note that for $\sigma_0 = 1.25$, the implication is that the EP's account for roughly half of the technical potential of a given module, with the remainder tied to unrepresented EP's or simply intrinsic to the module's role in the system. This makes a certain amount of intuitive sense and may be more easily justified as a modeling assumption than a more arbitrary division such as 60/40 or 83/17. However, without an understanding of the effects of different such proportions on the overall NOV results, little can be claimed beyond this intuitive appeal.

The space between the extremes of purely EP-based estimation and simply setting everything to 2.5 can be investigated by varying $\sigma_0 \in [0, 2.5]$. The effects of doing so for the Traffic Estimator designs are shown in Figure 7.5.

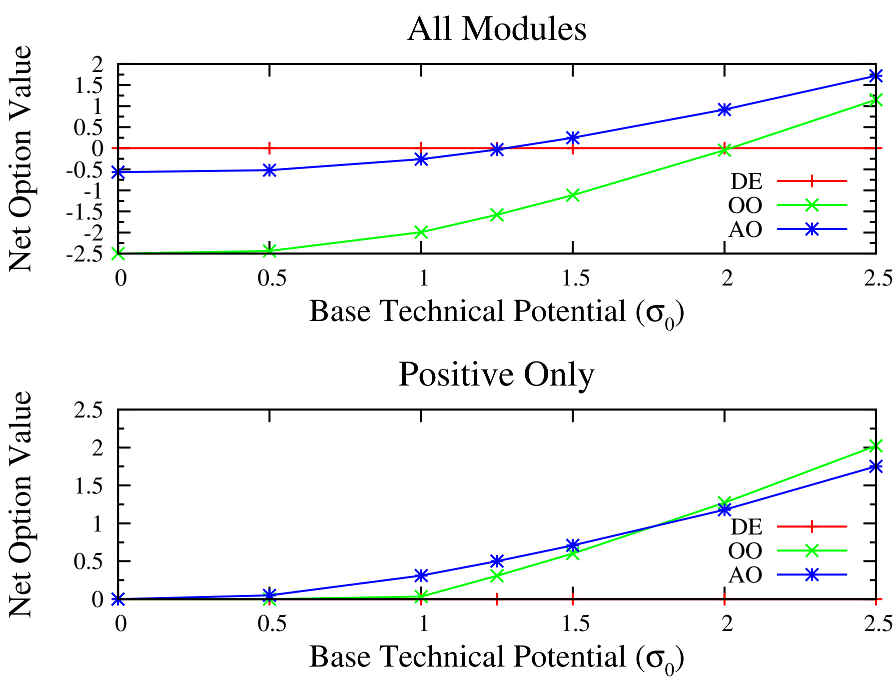


Figure 7.5: Effects of varying σ_0 on Traffic Estimator NOV Calculations

The flat red line for the DE version verifies that Equation 7.19 does not violate the "break-even" assumption for any reasonable value of σ_0. The most interesting trend is that the OO experiences more substantial gains in NOV, overtaking the AO design near $\sigma_0 = 1.75$ and finishing with a margin of 0.25 at $\sigma_0 = 2.5$. This is partially caused by a phenomenon known to the Aspect-Oriented community as "dependency inversion"[41]. In the OO design, the core algorithm is invisible to the rest of the system, so any contribution it might make to the NOV results is unhindered by visibility costs. In the AO design, however, the core algorithm is highly visible to the associated crosscutting interface, which offsets its ability to contribute positive value.

This is a valid phenomenon to expose in the overall results, and it starts to contribute positive option value for the OO design near $\sigma_0 = 1.0$ and ultimately contributes 0.26 to the NOV results for the OO design at $\sigma_0 = 2.5$. The rest of the gains over the AO design are due to the fact that the supplemental effects are implemented across five classes in the OO design, one for each effect, where the AO version concentrates them in four aspects, one for each datum. As σ_0 approaches 2.5, the "more modules is better" phenomenon begins to dominate, ultimately yielding a disparity on the order of 0.5 in favor of the OO design. This accounts for almost the entire margin gained over the AO design for $\sigma_0 > 1.5$ and suggests values of σ_0 larger than 1.5 may obscure important design features in this case study.

When combined with the minimum value that demonstrates the "dependency inversion" phenomenon, this implies a fairly narrow range of viable values for the base technical potential, centered on $\sigma_0 = 1.25$. As mentioned above, this value has a certain intuitive appeal, but it would be worthwhile to look at the relative effects on the other two components before adopting a specific value. Given the monotonic trends in Figure 7.5, it will be sufficient to evaluate the results for $\sigma_0 \in 0, 1.25, 2.5$ to determine whether the way that OO overtakes AO for the Traffic Estimator is an exception or a rule.

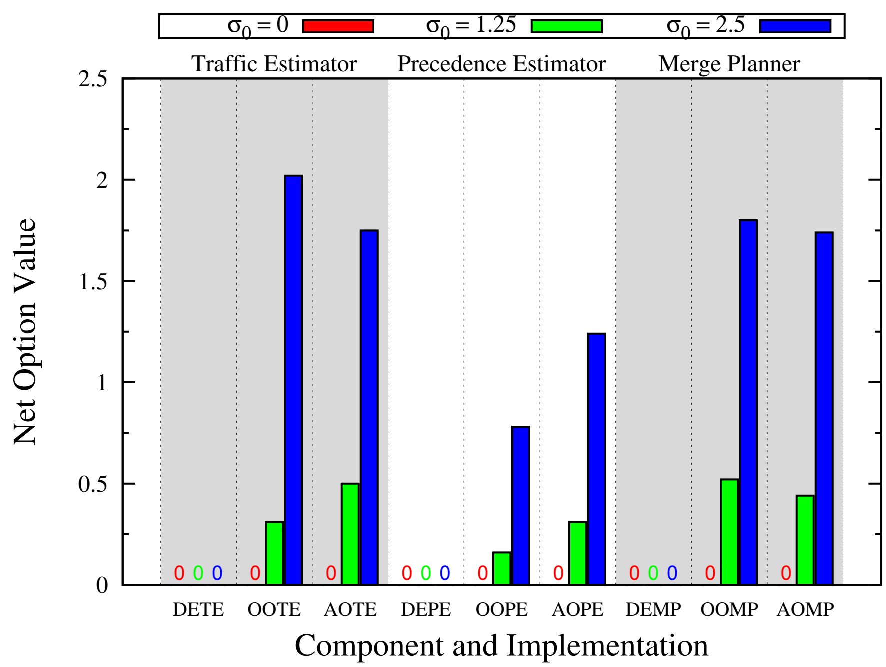


Figure 7.6: Effects of varying σ_0 on NOV results for all behavioral components, using *aggregation*.

The results of varying σ_0 for all three components are illustrated in Figure 7.6. As to consistency with previous results, all DE implementations have zero NOV, once again demonstrating maintenance of the "break-even" assumption. There are no positive NOV contributors for any component for $\sigma_0 = 0$, which reflects the results in Figure 7.4. Lastly, NOV monotonically increases in all cases for larger values of σ_0, which is consistent with the monotonic trends in Figure 7.5.

Curiously, as σ_0 goes from 1.25 to 2.5, neither the Precedence Estimator, nor the Merge Planner exhibits the same relative gains in the OO design as in the Traffic Estimator. Close inspection of the results for these components shows that both the "dependency inversion" and "more modules is better" phenomena are present in these two components, which begs explanation of their apparent absence in the overall results.

The nil net effect on the NOV results for the Merge Planner and Precedence Estimator is primarily accounted for by the XPI's in the AO designs for these components, which contribute counterintuitively large option values for $\sigma_0 = 2.5$. These effectively balance the dubious gains in the OO designs, but offsetting one set of questionable results with another is not a valid approach. Instead, a value of $\sigma_0 = 1.25$, which exposes interesting differences in the designs without exhibiting undesirable side-effects, will be adopted for further analysis.

Weighting Environment Parameters by Volatility

To this point, the estimation of technical potential has treated all EP's equally, which is consistent with previous work in NOV analysis, but is somewhat inconsistent with their use in this case study. As suggested in Chapter 2, some data on a robot will be more likely to trigger adaptation than others, and, intuitively speaking, dependency on more volatile data should yield higher technical potential for a given module. This can be represented by extending Equation 7.19 to weigh each EP differently, denoted by γ_e, and normalizing by the total weight, γ_{total}, to maintain the "break-even" assumption:

$$\sigma_i = \sigma_0 + (2.5 - \sigma_0)\Big(\sum_{i \text{ sees } e} \frac{\gamma_e}{\gamma_{\text{total}}}\Big) \tag{7.20}$$

In practice, γ_e can be unique per EP, which introduces several additional parameters to be estimated in the NOV formulation. However, these parameters represent an expectation of future change, or volatility, that can be more easily tied to external information than the other parameters to the NOV calculation. In the automotive industry, for example, the variations between different vehicle models may be known or even specified in advance, which would allow very concrete estimations of EP volatility. In this particular case study, the EP weights could be tied to known robot-specific details, such as the usage of particular sensors or traffic modeling techniques. Such advanced methods for estimating data volatility, and how they might feed back into the classification of primary vs. supplemental data, are under ongoing investigation.

For the purposes of this sensitivity analysis, a simple binary classification will be presumed: volatile and non-volatile data, which will be mapped to supplemental and primary data, respectively. Each type of data will be assigned a single weight, denoted γ_{pri} and γ_{supp}, and the overall weight will be normalized to one to preserve the "break-even" assumption from Section 7.2. Assuming that neither weight will be reduced to zero, the independent variable is the ratio of the two, which will be denoted $\alpha = \gamma_{supp}/\gamma_{pri}$. Larger values of α will thus weigh supplemental data more heavily, thus assigning more "volatility" to modules

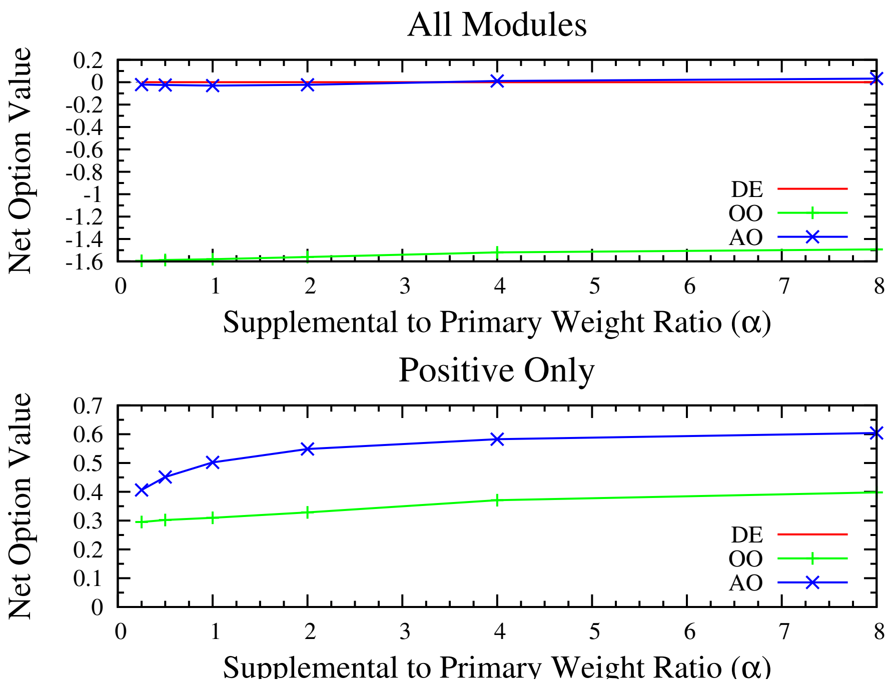


Figure 7.7: Effects of varying α on Traffic Estimator NOV results, using *aggregation* and $\sigma_0 = 1.25$.

that depend on those data. in Equation 7.20, and the effects of varying this ratio on the NOV results for the Traffic Estimator designs are shown in Figure 7.7.

Once again, the results for the DE implementation are all zero, demonstrating preservation of the break-even assumption across all values of $\alpha \in [0.25, 8.0]$. The ordering between the AO and OO designs is also preserved across the same span, so the net effect may simply be an adjustment of the contrast between AO and OO implementations.

In both the AO or OO designs, the modules that depend on supplemental data experience an intuitive increase in NOV, and the core algorithms, which depend only on primary data, experience a corresponding decrease. The overall effects almost cancel each other out, as indicated by the nearly horizontal results in Figure 7.7(top).

For the "true" NOV summation in Figure 7.7(bottom), the contrast between the AO and OO designs is significantly enhanced up to and including $\alpha = 2.0$, but largely parallel growth thereafter indicates that values beyond this will not significantly impact the final results. As such, Figure 7.8 evaluates the NOV results for all three components for $\alpha \in \{1.0, 2.0\}$.

The effects on the other two components, shown in Figure 7.8, are largely consistent with those for the Traffic Estimator, namely that the AO designs experience a substantial boost in NOV for $\alpha = 2.0$, where the OO designs do not. The most notable demonstration of this is the Precedence Estimator, whose OO design significantly lost option value in this context. For $\alpha = 1.0$, the core algorithm of the Precedence Estimator contributed a small amount of positive option value, per the "dependency inversion" phenomenon discussed above. This contribution was significantly reduced for $\alpha = 2.0$, and the corresponding gains in the two supplemental effect classes, which are depressed by visibility to the delegate interfaces, were

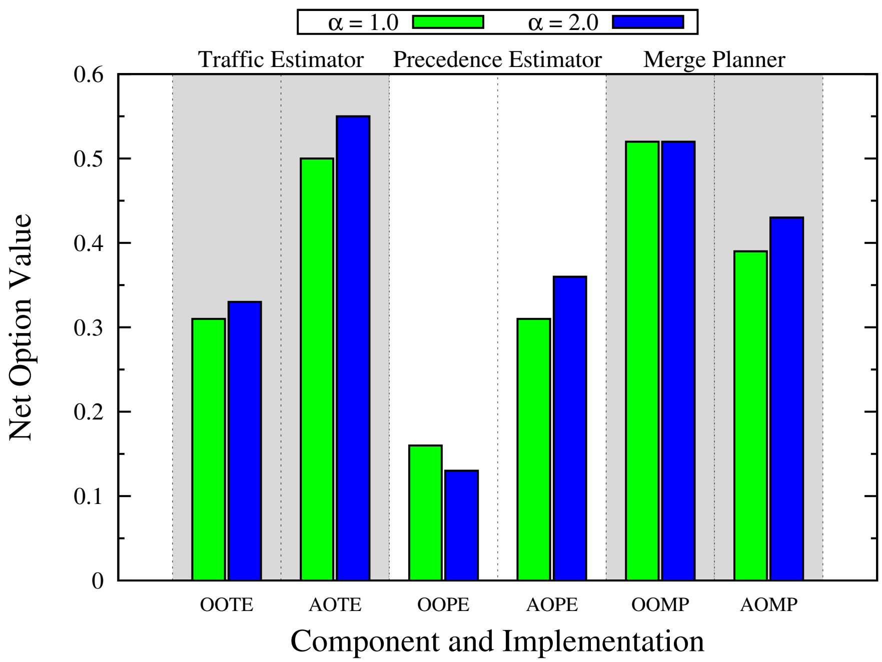


Figure 7.8: Effects of varying Supplemental to Primary Weight Ratio on NOV results for OO and AO implementations of all behavioral components.

not enough to compensate. To a certain extent, this represents an enhancement of the NOV formulation's ability to model the value of the "obliviousness" property of aspects, which is the flip-side of "dependency inversion" and encourages the use of $\alpha = 2.0$ in further analyses. Moreover, a ratio near 2:1 is intuitively appealing in that supplemental data can be seen as *at least* twice as likely to trigger adaptation as primary data in this context.

Still, the visibility costs in the OO design warrant closer inspection, as the delegate interfaces that impose those costs are, in reality, quite small and concise. In the above case of the Precedence Estimator, these modules were assigned 22% of the total system complexity when, in fact, they accounted for less than 9% of the source code for the OO implementation. The following section explores the significance of this disparity by incorporating source code size into to the determination of module complexity.

Weighting Complexity by Lines of Code

The final variation explored in this chapter is an alternate allocation of complexity according to the source code size instead of simply by DP count as in Equation 7.11. As discussed above, the DP's were selected to represent either single classes or a small collections of methods, so it was straightforward to count the lines of code that contributed to each. A weighted sum will be used to explore the sensitivity of the results, with the weight given to the LOC measurement denoted β in:

$$\eta_i = \beta\Big(\frac{LOC_i}{LOC_{total}}\Big) + (1 - \beta)\Big(\frac{M_i}{N}\Big) \tag{7.21}$$

M_i and N are the count of DP's in module i and in the whole design, respectively, from Equation 7.11. The results of varying β from 0 to 1 for the Traffic Estimator designs are presented in Figure 7.9.

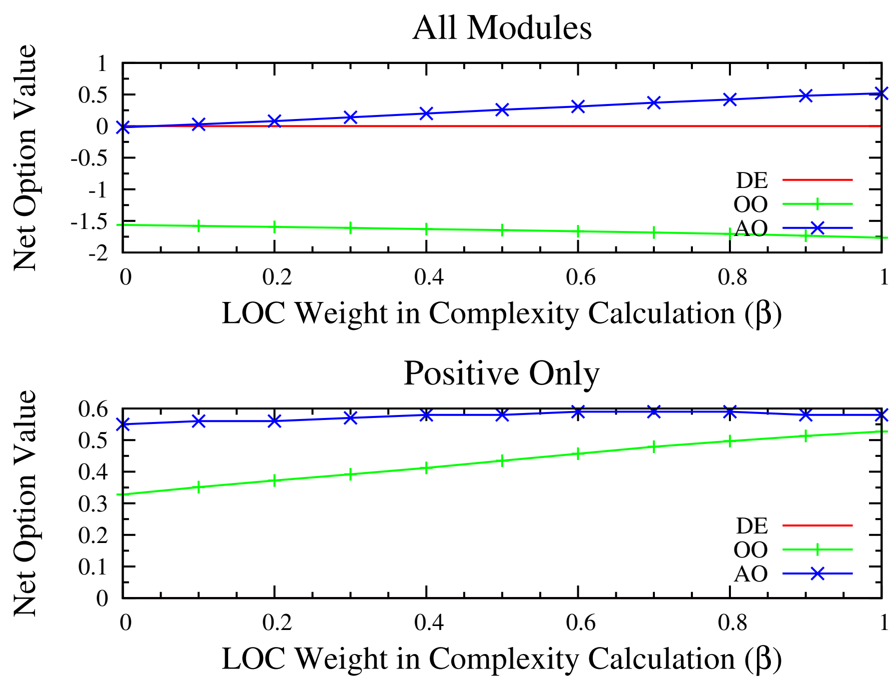


Figure 7.9: Effects increasing the weight, β, given to LOC in the complexity model for Traffic Estimator NOV Calculations, using *aggregation*, $\sigma_0 = 1.25$, and $\alpha = 2.0$.

The overall effect of introducing the LOC measure is to concentrate the estimated complexity into the core algorithm, which represents 71% of the source in the OO design and 84% for the AO design. As shown in Figure 7.9(top), this causes a significant increase in overall NOV for the AO design, but has a slight negative impact on the OO design. This is an enhancement of the "obliviousness" effect discussed above, where the OO delegate interfaces are visible to the core algorithm, whose increased complexity imposes a higher visibility cost, depressing the overall NOV results.

Curiously, this effect is absent in the "true" NOV calculation, which indicates that the "losses" in the OO design were primarily in modules that already had little or no value. Instead, the results for both the AO and OO designs are somewhat increased by the LOC allocation of complexity, with the OO design experiencing much higher gains and narrowing the gap to the AO design by roughly 75%. This is also caused by the concentration of complexity in the core algorithm, but where it increases the visibility costs imposed on the delegate interfaces discussed above, it also reduces the visibility costs that these interfaces impose on their client modules, which encode the supplemental effects. In a sense, this restores a certain amount of equability to the NOV model's treatment of "obliviousness", in that the costs associated with "non-obliviousness" are reduced to more reasonable levels. This alone encourages at least a partial contribution of LOC to the overall complexity metric.

The last, and perhaps most noteworthy, effect of LOC-based complexity estimation is that the NOV results for the AO design are, if only subtly, non-monotonic in Figure 7.9(bottom). That is, they reach a local maximum near $\beta = 0.8$ and fall slightly again as $\beta \to 1.0$. Closer inspection shows this to be a sensitivity of the first two elements of the NOV formulation to very small values of complexity. In this case, the slight shift in complexity as $\beta \to 1.0$ caused the NOV curves for the AO "supplemental effect" modules to crest for larger values of k but also at a slightly lower option value than for $\beta = 0.8$.

This effect, which can be thought of as a trivialization of the contribution of these modules, is also present in the OO design, but it is countered by the reductions in visibility cost discussed above. While this suggests that using LOC as the absolute measure of complexity will yield inaccurate results, there are still interesting issues that may be had by partial application, such as for $\beta = 0.5$. To evaluate the effects on the other components, Figure 7.10 plots the NOV results for all three components for $\beta \in 0.0, 0.5, 1.0$.

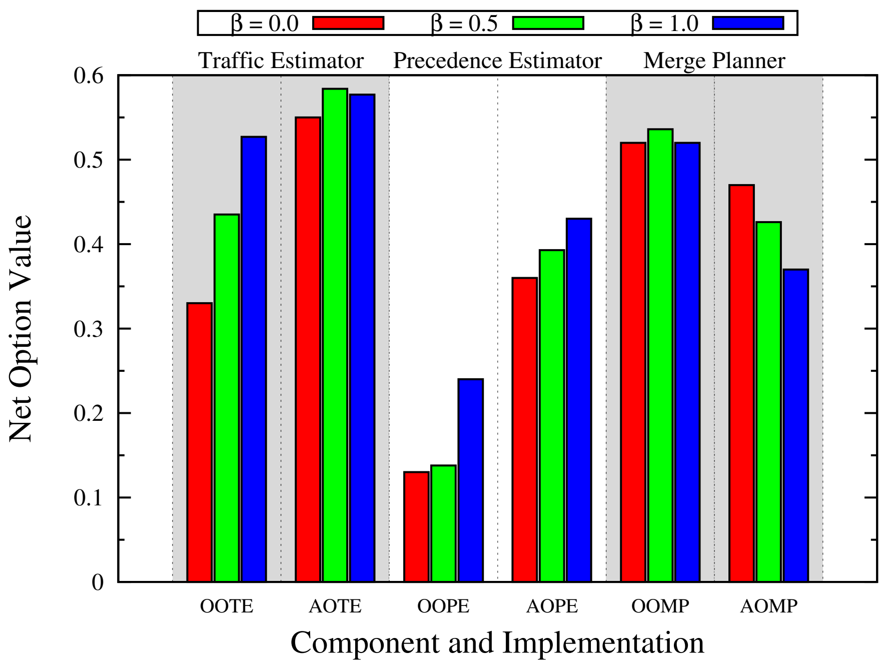


Figure 7.10: Effects of using lines-of-code (LOC) to allocate complexity for all behavioral components, using *aggregation*, $\sigma_0 = 1.25$, and $\alpha = 2.0$.

For all three components in Figure 7.10, the NOV of the OO design improves somewhat relative to the AO design for $\beta = 0.5$, and much more drastically for $\beta = 1.0$, which is largely consistent with the detailed discussion of the Traffic Estimator's results. The exception to this is the substantial reduction in NOV for the AO design of the Merge Planner, both for $\beta = 0.5$ and $\beta = 1.0$. This is an exacerbation of the "trivializing" effect discussed above, induced by the sheer magnitude of the Merge Planner's core algorithm, which accounts for 94% of the source code in the AO implementation. Even in the OO design, this effect overtakes the reduction in visibility cost as $\beta \to 1.0$, as indicated by the slight drop in the "OOMP" results in Figure 7.10.

This issue might be resolved by selecting alternate values for β, but it seems more likely that the use of LOC as a complexity measure may controvert one of the original simplifying assumptions in Section 7.2, where the arbitrary cost function $C_i(\eta_i)$ is replaced with simple proportionality. In this case, setting $\beta = 1$ in Equation 7.21 amounts to assuming that costs are directly proportional[4] to code size.

For the purposes of this analysis, the issues discussed above will be alleviated by adopting a value of $\beta = 0.5$, effectively taking the average of the DP- and LOC-based complexity estimates. As a possible path of future research, it would be worth investigating the relative impact of other, more industrial-strength measurements of complexity, either to be used directly in the NOV formulation, or else to help identify the appropriate granularity of DP's that will allow the simplifying assumptions from Section 7.2 to more accurately model the expected costs of experimentation and integration.

Final Results

At this point, it is informative to collect all of the enhancements and compare the final results to the initial results, as illustrated in Figure 7.11:

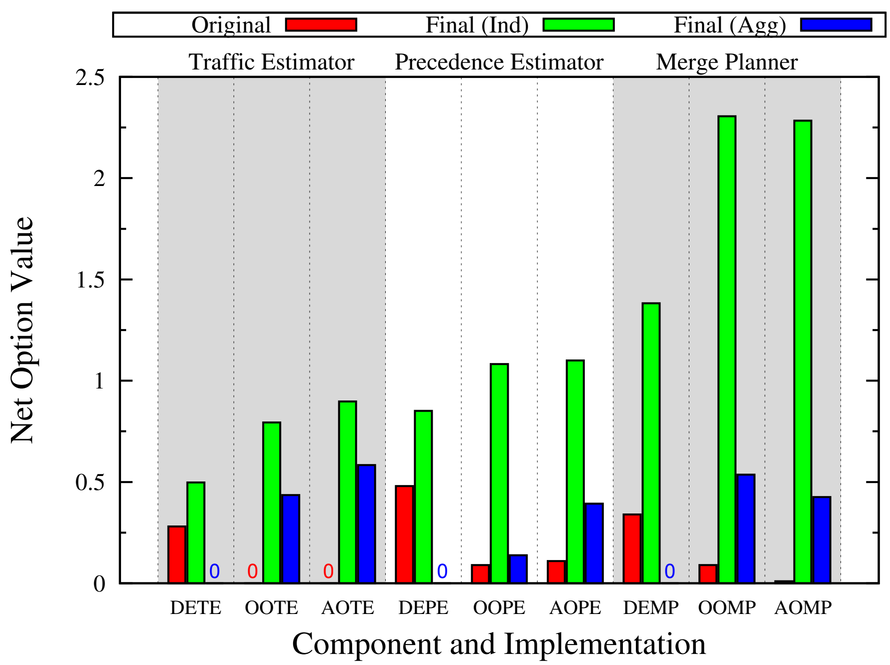


Figure 7.11: Comparison of initial NOV results (red) to final results, both under aggregation (blue) and with each DP treated an individual module (green), otherwise using $\sigma_0 = 1.25$, $\alpha = 2.0$, and $\beta = 0.5$.

It is worth noting that the final results (blue bars) are all on the same order of magnitude as the original results (red bars), with values ranging roughly between 0.0 and 0.5. This indicates that the alternate parameter estimation techniques do not arbitrarily inflate the

[4]While the correctness of "direct proportionality" is certainly debatable, source code size, in being one of the few directly measurable properties of a software component, is a widely used and accepted proxy for development costs.

NOV results, lending credence to their being more accurate models of the costs, benefits, and overall value of these designs.

Second, wherever there was an ordering between the AO and OO designs to begin with, that ordering is preserved, with the contrast somewhat enhanced in the final results. This is true both when the AO design exceeds the OO design, as in the Precedence Estimator, and vice-versa, as in the Merge Planner, indicating that the parameter estimates do not unfairly favor one of those approaches over another.

However, the same cannot be said as easily for the DE implementations, as the combined effects of the new parameter estimates are all hidden behind the "break even" assumption from Section 7.2. This holds the NOV for the DE versions at zero for all experiments after the introduction of aggregation at the beginning of this section. The green bars in Figure 7.11 help clarify this matter by removing aggregation, but retaining the other three enhancements discussed above.

These results are both highly internally consistent, in that all implementations experience a significant boost in NOV, and also consistent with the results that include aggregation, as the NOV ordering between designs remains fixed whether the DP's are aggregated into larger modules or not. Both the AO and OO designs yield higher NOV than the original implementation, and the better of the AO and OO designs is retained as before, if only by a narrow margin. This consistency across granularity is reassuring of the validity of the final results, and suggests another interesting path of research into parameter estimation techniques that preserve, or at least predictably affect, NOV results across many levels of granularity.

7.4 Discussion

The simplified NOV formulation used in previous work yielded results that suggested that the proposed methodology of separating supplemental effects from core algorithms was not particularly valuable. However, close analysis of the validity of the underlying assumptions, and consequent application of alternate parameter estimation techniques yielded results that instead favor the proposed AO and OO approaches over the original, monolithic implementation.

In order to avoid the perception that the parameters have simply been manipulated to yield the desired results, each incremental variation in parameter estimation was tied to previous work, or at least previous suggestions, in the NOV literature. Where possible, these variations were framed as weighted averages against the original parameter estimation techniques, the corresponding sensitivity of the NOV results was analyzed, and the underlying causes were discussed as part of justifying their application to the final formulation. This sensitivity analysis is, to the authors' knowledge, a novel contribution to the field, providing useful insights into the nature and behavior of the NOV model and suggesting several possible paths of future research. These include further variations on parameter estimation techniques , such as incorporating a more accurate models of volatility or end-user value, and also a better understanding of the impact of granularity on the overall NOV results.

As to the original design problem of separating robotic algorithms according to dependencies on primary vs. supplemental data, the final results from Figure 7.11, backed by the preceding sensitivity analysis and discussion, indicate that the overall methodology can provide significant value in the context of robotic software. Whether AO or OO techniques are specifically better-suited to this task is somewhat less clear, but it seems both likely and intuitive that some platform-specific enhancements will lend themselves more naturally to one implementation technique or another. Ultimately, both techniques add substantial value to the design of these components, encouraging future designers to explore the modular treatment of primary and supplemental effects according to the most effective implementation strategy for their system.

As partial validation of these results, the next chapter discusses the extension of the refactored urban driving artifacts to accommodate *additional* data. This data, drawn from work on vehicle-to-vehicle (V2V) communications standards, can have significant effects on almost any autonomous driving algorithm, and the implications for the components in this case study yield interesting insights into the value of the proposed *primary* vs. *supplemental* methodology.

Chapter 8

Extension to Novel Input Data

The work detailed to this point focuses on identifying, refactoring, and analyzing existing supplemental effects in autonomous driving software. The results of this work indicate that those existing effects are well encapsulated by the AO and OO design techniques, and that the corresponding artifacts are more adaptable than the original, direct encoding relative to *alteration* or *absence* of individual supplemental data.

The remaining problem, of a design's ability to accommodate *additional* data through modular augmentation, is the focus of this chapter, beginning in Section 8.1 with the identification of a problem domain that can yield data and candidate effects well beyond the scope of the existing contents of the `MovingObstacle` representation. Section 8.2 then draws several examples from this domain and suggests several compelling effects that may be applied to the autonomous driving components discussed in Chapter 5. These effects are elaborated upon to yield detailed functional requirements, which are discussed along with the associated adaptations they would compel for each design of each component in Section 8.3. The critical results of this chapter are summarized in Section 8.4 before proceeding to the complementary case study in Chapter 9.

8.1 Introduction: Novel Supplements for Autonomous Driving

As discussed in Chapter 2, there are many and various reasons why *additional* data may be made available, such as adding new sensors to an existing platform, incorporating new perception algorithms on an otherwise unchanged platform, or simply porting software components to a completely different platform. It follows that, as opposed to contriving possible *additional* data out of "thin air", it would be more compelling to identify a specific change in the surrounding system, and derive candidate *additional* data and *supplemental effects* from the consequences of that change.

One particularly interesting possibility arises from a common criticism of the Urban Challenge software system: that the sensing and computing resources that were used on Boss are too costly to consider for inclusion in consumer automobiles. That is, the cost constraints of consumer automobiles restrict both the suite of sensors that may be used, and the computing resources available to process the resulting data, to the point that the purely sensor-based techniques employed for the Urban Challenge are not feasible in a production setting.

As a complement to these limited sensing and computing resources, the automotive industry has proposed the use of explicit communications, both from vehicle to vehicle (V2V) and between vehicles and the surrounding infrastructure[1] (V2I/I2V). Such communications were expressly forbidden by the Urban Challenge rules, so the incorporation of such V2V information is both a logical and compelling change that may occur as part of ongoing research with Boss, or in the transition of algorithms deployed on Boss into a more production-relevant setting.

Most importantly, the Society for Automotive Engineers (SAE), in conjunction with the National Highway Traffic Safety Administration (NHTSA), are working on a nationwide standard, analogous to JAUS[27] for military robots, for the content and semantics of V2V and I2V messages. This standard, the "Dedicated Short Range Communications (DSRC) Message Set Dictionary"[15], is a testament to the insight and experience of its contributors, as it covers a very wide variety of vehicle capabilities and traffic situations to an exquisite level of detail. Message specifications include vehicle status messages, warnings for impending collisions, notifications of approaching emergency vehicles, identification of work zones, and many other traffic-related information that could influence any number of autonomous driving algorithms. These messages are organized into:

- **Messages** (or message sets), which are the top-level elements that are actually transmitted between vehicles and/or the surrounding infrastructure. These are analogous to message classes, such as the `MovingObstacle` representation, in the Tartan Racing system.

- **Data Frames**, which are smaller groups of closely-related data. These are analogous to compound types, such as poses, waypoints, etc., in previous discussion.

- **Data Elements**, which are semantically-bound primitive types, such as integers and enumerated types that represent lengths, speeds, status bits, etc.

There is a partial compositional ordering between these three elements, in that Data Frames are comprised of Data Elements, and Messages can contain both Data Frames and Data Elements. This is consistent with message compositions in the Tartan Racing system, so it would be reasonable to model hypothetical V2V introductions as *additional* member data in various messages used in the existing system. In particular, the `MSG_BasicSafetyMessage` definition, which *must* be transmitted in some form by all V2V-capable vehicles, includes many data that are already present in the `MovingObstacle` representation, such as position and velocity of the transmitting vehicle, that could be used to augment (or supplant) existing sensing and modeling techniques currently employed on Boss.

The qualification "in some form" alludes to the fact that there is some *basic* form of the safety message that must be supported by all vehicles, but there are also many so-called "optional" data that may be appended to this basic message to convey *additional* information, such as acceleration, turn signal state, or even driver identification, as desired.

[1]A common example of I2V communications would be a traffic signal reporting its state to nearby vehicles, relieving those vehicles of the need to identify the traffic signal state using machine vision or other techniques.

These "optional" data provide a very rich set of candidates for *supplemental* treatment, as their availability may actually vary from one vehicle to the next, or even one message to the next, as an exacerbated form of the "run-time" adaptations discussed in Chapter 2. Several such examples of "optional" data, along with the "required" data in the basic safety message, are discussed in the next section, focusing on how they may be incorporated into the `MovingObstacle` representation, along with a high-level discussion of the autonomous behaviors they could enable. This is followed by a more detailed presentation and analysis of several candidate supplemental effects in Section 8.3.

8.2 Moving Obstacles from V2V Data

The Basic Safety Message

The Basic Safety Message (BSM) is expected to be transmitted[2] by all V2V-capable vehicles, so it is reasonable to expect the contents of this message to be readily available for a subset of the surrounding traffic. Although this and other messages are specified in [15] at the level of individual byte arrays and bit fields, for the purposes of this discussion, an analogous UML representation, shown in Figure 8.1, will be used instead.

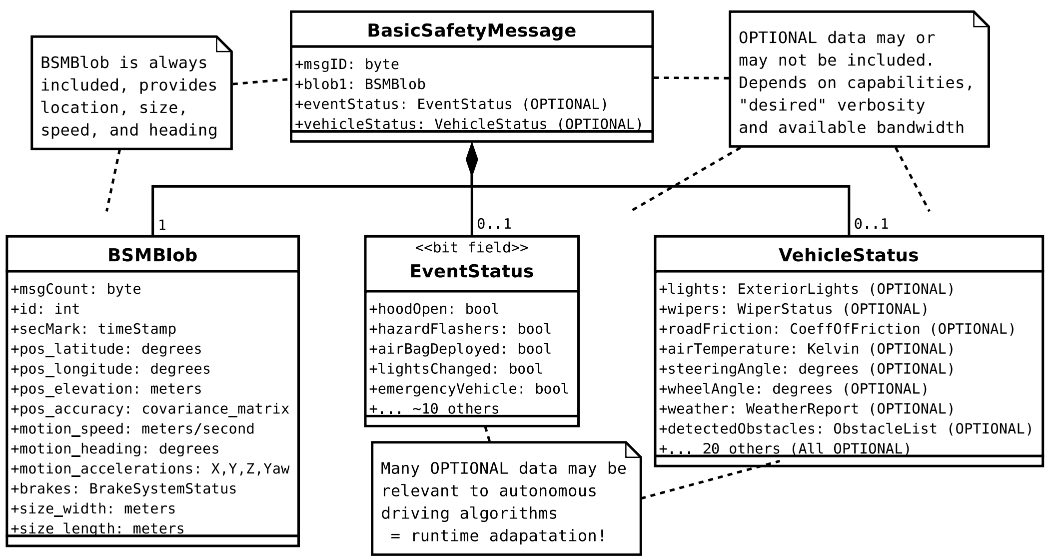


Figure 8.1: UML representation of the DSRC Basic Safety Message

As of January 2009, the Basic Safety Message, represented by the `BasicSafetyMessage` class in Figure 8.1, consisted of:

- A message ID, which is the standard first byte of any DSRC message;
- An efficiently-packed "binary blob" of basic status data, represented by the `BSMBlob` class;

[2] In fact, the BSM is often referred to as a "heartbeat" message, and is expected to be transmitted periodically, as often as every 10ms.

- An "optional" bit-mask of critical event flags, represented by the `EventFlags` class;
- An "optional" verbose vehicle status structure, represented by the `VehicleStatus` class;
- An "optional" body of custom or vendor-specific data, called the "local" basic safety message, and excluded from Figure 8.1 for clarity.

The only data that *must* be transmitted as part of a BSM is the so-called `BSMBlob`, where all other members in all other classes depicted in Figure 8.1 are "optional". That is, their inclusion is at the discretion of the transmitting vehicle. Even still, the basic vehicle status "blob" contains a great deal of useful geometric information about the transmitting vehicle, including position, heading, speed, acceleration, and size information that are clearly relevant to a wide variety of autonomous driving algorithms. In fact, as discussed in more detail below, the contents of the "blob" are sufficient to populate the *primary* data in the `MovingObstacle` representation, suggesting a strong similarity between the *primary* vs. *supplemental* methodology proposed by this thesis, and the decision as to "required" vs. "optional" data in the DSRC specification.

The first "optional" datum is the `EventFlags` bit-mask, which conveys a variety of boolean states of the transmitting vehicle, ranging from mere curiosities, such as `hoodOpen` or `wiperStatusChanged`, to more critical issues such as `airBagDeployed` or `emergencyVehicle`. There are more than a dozen such status bits, many of which could be relevant to the autonomous driving algorithms discussed in this work. For example, the indication of `hazardFlashers` might be used to cull an "irrelevant" vehicle from an intersection or to trigger more aggressive passing maneuvers around disabled vehicles.

The second "optional" datum is the verbose `VehicleStatus`, which is, in turn, composed entirely of "optional" data fields, ranging from straightforward information about steering angle and external indicator state to more esoteric data about perceived road friction, air temperature, or weather reports. There are nearly thirty such "optional" data in the `VehicleStatus` structure, some of which even supersede the contents of the "binary blob" by providing more accurate measurements of velocity, position, etc., along with more extensive representations of error or uncertainty thereof. Only a subset of these "optional" data are presented in Figure 8.1, focusing in particular on those that would be most relevant to the autonomous driving algorithms under investigation. Among these, the `ExteriorLights` data, which includes representations for brake, hazard, and turn signals, could be used to implement a wide variety of cooperative or social driving behaviors, such as opening up a merge gap for a signalling vehicle in an adjacent lane, or disallowing passing maneuvers on the same side as an actively-signalled turn.

Integration into the MovingObstacle Representation

Figure 8.2 highlights the fact that the contents of the Basic Safety Message could easily influence the derivation of every member of the existing `MovingObstacle` representation. In fact, as mentioned above, the primary data may be completely populated via unit and/or coordinate-frame conversion of the contents of the basic status "blob":

- `MovingObstacle::pose` may be derived from `BSMBlob::pos_{latitude,longitude}`, along with `BSMBlob::motion_heading`;

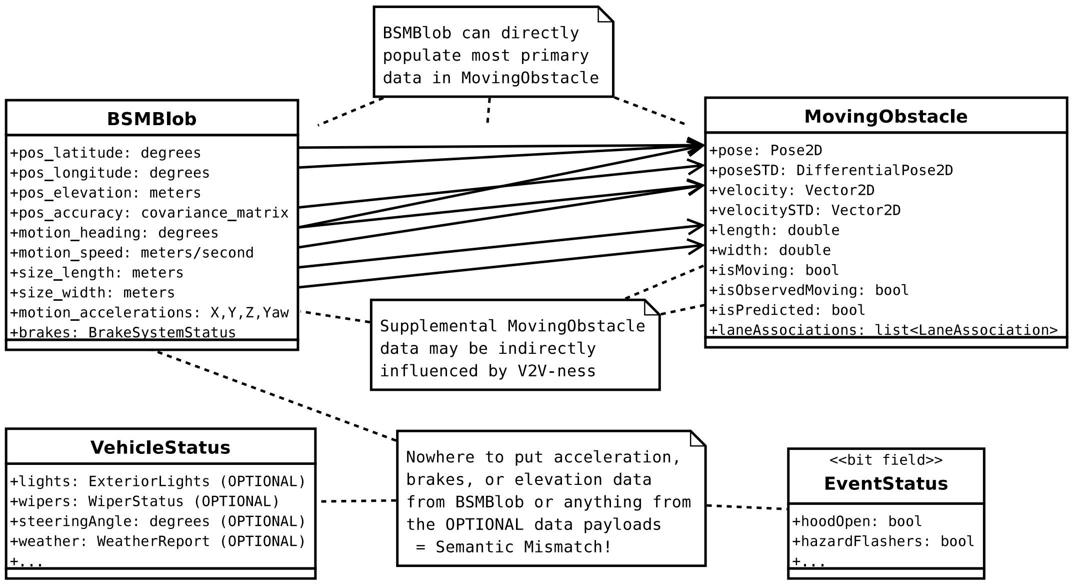


Figure 8.2: Populating a `MovingObstacle` from the contents of a condensed representation of the DSRC `BasicSafetyMessage`

- `MovingObstacle::velocity` may be derived from the contents of `BSMBlob::motion_{heading, speed}`;
- `MovingObstacle::length,width`, may be derived from the homonymous data in `BSMBlob::size_{length,width}`;

When these data are populated in this manner, it is likely that they may be "trusted" more than the same data derived from sensors such as RADAR or vision. It follows that the "V2V-ness" of an obstacle will affect the derivation of the supplemental `MovingObstacle` data as well, such as:

- `IsMoving` may be simplified to a threshold on `BSMBlob::motion_speed`;
- `IsObservedMoving` may follow `IsMoving` more closely given the "extra trust" placed in V2V data, although the historical motion of a vehicle may still be checked for consistency with "nominal" traffic behavior.
- `LaneAssociations`, like `IsObservedMoving`, may be more responsive given the more accurate geometric information, and may also be enhanced by professed accuracies (per `BSMBlob::pos_accuracy`).
- `IsPredicted` would typically be *false*, or could be tied to some "staleness" threshold relative to the last receipt of a V2V message from that particular vehicle. There may also be alternate occlusion or failure models for the V2V signal that may be hidden behind this flag.

While these suggest the possibility of *alterations* to the semantics of these existing supplemental data, it is not clear that the semantic shift will be sufficient to warrant modification of the existing supplemental effects. It is more likely, however, that some of these supplemental data would be excised in a transition to a production setting, especially the `IsPredicted` and `IsObservedMoving` flags, which are closely bound to Boss- and Urban Challenge-specific assumptions about occlusion and traffic behavior, respectively.

In any case, these effects are already encapsulated by the AO and OO designs presented in Chapter 5, and the analysis in Chapters 6 and 7 indicates that the *absence* or *alteration* of data are well handled by the proposed methodology. The focus of this chapter is the accommodation of *additional* data, and there are many candidates for *additional* treatment highlighted in Figure 8.2. That is, there are several data in the `BSMBlob` representation, along with a wide variety of other "optional" data, as discussed above, that have no means of expression in the existing `MovingObstacle` class.

A complete evaluation of all such "optional" data, even restricted to one or two candidate effects per datum, would be intractable in the scope of this thesis. Instead, this work will explore a wide variety of candidate effects, based on comparatively few data from the BSM representation, that are specifically selected to exercise the adaptation interfaces generated as part of the refactoring experiments in Chapter 5. Of all the V2V data above, a reasonably compelling set of example supplemental effects can be derived by incorporating two comparatively simple members into the `MovingObstacle` class.

- A boolean indication of whether or not a given obstacle is backed by V2V messages, which, as mentioned above, would convey a certain sense of "extra trust" that can be placed in the *primary* data.

- An enumeration of external signal state, which, at a minimum, could be derived from the `BrakingState` member of `BSMBlob`, but may also incorporate optional data such as the `hazardFlashers` event state or may be directly derived from the "optional" `ExteriorLights` member of `VehicleStatus`. Regardless of the specific source, the external signal state can be used to infer the "intent" of a given vehicle and enact various social or cooperative behaviors, such as creating merge windows or restricting passing maneuvers, as discussed above.

HasV2V

A simple boolean indication that a Moving Obstacle is backed by V2V data, such as `MovingObstacle::hasV2V`, would imply the receipt of at least the minimal content of `MSG_BasicSafetyMessage`, which in turn implies the aforementioned "extra trust" in the *primary* pose, size and velocity data. This `hasV2V` flag could influence autonomous driving algorithms in much the same way, and in many of the same places, as the original `isMoving` and `isObservedMoving` states, including:

- The determination of whether or not a candidate obstacle is a "false positive", or whether it is otherwise relevant to a given situation;

- The determination of various thresholds, such as maximum range at which an obstacle may be included in merge or yield calculations, or how far away the obstacle can be from the centerline and still be "in" a given lane.

This can also affect "higher-level" policies, especially by reducing the degree of "conservativeness" applied in various driving algorithms, such as:

- The estimation of the distance to the lead vehicle, which may assume a "slightly closer" obstacle in uncertain situations;
- The estimation of "worst-case" distances and velocities for distance-keeping, merge or yield calculations when faced with similar ambiguities;
- The determination of the desired distance-keeping gap, which is currently tuned for ultra-conservative behavior, but may be reduced if the lead vehicle "hasV2V" to yield a sort of convoy behavior;

The "V2V-ness" of an obstacle may also affect more algorithm-specific policies, such as the Merge Planner's determination (**MP.D.11**) of whether or not an obstacle should be granted a "courtesy gap" when considering a front-merge. This is one way that the methodology proposed by this thesis enhances the adaptability of robotic algorithms: by providing a coherent and explicit enumeration of such algorithm-specific policies through a dedicated interface that allows them to be "tweaked" according to novel inputs. That way, a developer who is considering candidate effects of some *additional* data may begin by perusing the existing adaptation interface, instead of delving directly into the underlying source code, and considering how the new data might influence each exposed point of adaptability. There is always the possibility, however, that the existing adaptation interface is not sufficiently expressive to capture the desired effects of some novel input data, the ramifications of which are discussed in detail in Section 8.3, especially relative to advanced cooperative behaviors that might be enabled by turn-signal status information.

TurnSignalState

The second supplemental datum that will be introduced into the `MovingObstacle` class is a representation of the obstacle's turn signals, such as shown in Listing 8.1.

```
enum MovingObstacle::TurnSignalState {
  Unknown = -1,            // accommodates absent and/or minimal V2V
  InActive = 0,            // conveys both "known" and "inactive"
  BrakeLights = 0x01,      // vehicle is braking
  LeftTurnSignal = 0x02,   // left turn
  RightTurnSignal = 0x04,  // right turn
  HazardSignal = 0x06      // (LeftTurnSignal | RightTurnSignal)
  // ... room for others, if desired
};
```

Listing 8.1: Example turn-signal enumeration for incorporation into the `MovingObstacle` Representation

The turn signal state is arranged as an enumerated bit-field to represent the possibility of combined states, such as "braking" and "left-turn", and also that certain combinations are not valid, such as signalling right- and left-turns simultaneously, which is illogical and otherwise indistinguishable from the "hazard-flasher" state[3]. This is consistent with how the optional `ExteriorLights` data are represented in [15], except that the `TurnSignalState` enumeration listed above excludes information about high beams, fog lights, etc., that are less relevant to autonomous driving algorithms.

Even restricted to the four signals enumerated in Listing 8.1, there is a great deal that can be inferred about "driver intent" that could be incorporated into a wide variety of algorithms, such as:

- Increasing the distance-keeping gap for active turn signals in order to accommodate the typical reduction of velocity that precedes an "upcoming" turn, or

- Using the `HazardSignal` state to influence various thresholds, timeouts and/or explicit policies for ignoring and/or passing stopped traffic.

Turn-signal data could also be used to generate completely new policies, including cooperative driving behaviors such as creating a merge gap for a vehicle in a neighboring lane, or disallowing passing maneuvers around turning traffic, as discussed above. In the current implementation, many of these behaviors are implemented outside of the three components discussed so far, especially the "Lane Selector" and "Distance Keeper" classes shown in Figure 5.1. Forwarding the supplemental `TurnSignalState` to these components to enable such alternate policies requires higher-level architectural changes, along with refactoring and generation of adaptation interfaces for these two additional components. Candidate designs for these additional components and the associated architectural changes are discussed toward the end of Section 8.3.

A Note About Acceleration Data

To this point, the acceleration data included in the `BSMBlob` representation in Figure 8.1 has been largely ignored as a candidate for supplemental treatment, even though the most basic V2V messaging would make it available as *additional* data. This requires a certain amount of explanation, as acceleration could easily influence many detailed calculations in any number of autonomous driving algorithms.

From a practical standpoint, virtually none of these calculations were exposed in the adaptation interfaces derived in Chapter 5, so, no matter what specific effects are desired, the inclusion of acceleration data would entail significant work in the *core* implementation of the Traffic Estimator, Precedence Estimator, Merge Planner, and possibly other components. While it would be possible to expose the necessary calculations through augmented adaptation interfaces, the application of acceleration data would have to completely supplant the underlying calculation, as was done for the `LaneAssociations` list for **TE.S.4** and **MP.S.1**. In this case, however, the "new" functionality would largely be a replication of the "underlying" functionality, with acceleration (and $time^2$) terms interleaved at the appropriate places. Such replication is well recognized as an artifact of poor design choices, as it forces two mostly-identical segments of code to be manually synchronized. This makes acceleration data a poor candidate for *supplemental* treatment.

[3]Not to mention the fact that it is not generally possible to simultaneously signal both a right- and left-turn using standard vehicle controls.

From another perspective, acceleration data also fits many of the intrinsic criteria for *primary* data discussed in Chapter 2. That is, acceleration data is measured in real units (G's, or m/s^2), does not require any significant "interpretation", and can thus be taken at "face value" by consuming algorithms. Moreover, there was a system-wide mandate to assume zero acceleration in all Tartan Racing software components, which provides a meaningful "default" value, which was the final condition for data that *may*, if not *should* be treated as *primary* data.

For these reasons, the methodology proposed by this thesis would suggest *primary* treatment for the acceleration data, and specify the default value of zero when that data is unavailable[4]. Interestingly, the context-specific usefulness or validity of the acceleration values could be determined according to related supplemental data, as was done for velocity data using `isMoving` and `isObservedMoving`, but in this case using the `hasV2V` flag, discussed above. Still, acceleration would itself be counted among the primary data in the `MovingObstacle` representation, and will not be further pursued in this discussion, which will focus on the candidate effects of `hasV2V` and `TurnSignalState`, described above.

These effects are described as formal functional requirements in the next section, which also presents the corresponding implications for each design of the Traffic Estimator, Precedence Estimator and Merge Planner. Particular emphasis is placed on how well the adaptation interfaces generated in Chapter 5 accommodate the necessary changes, what extensions to those interfaces would have to be performed, and what might be done to make them more generally expressive for future adaptations, which are the principal results of this chapter.

8.3 V2V Effects on Autonomous Driving Algorithms

HasV2V: False-Positive Culling

The most direct application of the proposed `hasV2V` flag is to ensure that obstacles supported by V2V transmissions are treated as cars, and not ignored, for instance, as an arbitrary road blockage or roadside vegetation:

CX.1 Candidate obstacles supported by V2V signals, as indicated by `MovingObstacle::hasV2V`, shall never be ignored as an intrinsic false-positive in any behavioral context.

In many ways, this usage of `hasV2V` reflects an "ultimate trust" in the existence of the obstacle, where other supplemental data, such as "moving" or "observed-moving" could only convey "degrees" or "facets" of trust in specific instances. It is thus likely that the adaptations implied by **CX.1** will be well handled by the existing adaptation interfaces.

However, the question of situational "relevance" is not exactly the same as the idea of "is definitely a car", so the implementation of this requirement will not necessarily be a blanket override of all related supplemental effects. This is the source of "intrinsic" in the requirement phrasing, as an obstacle that is "definitely a car" may also be "definitely too far away to matter". It follows that careful analysis of the existing XPI/Delegation interfaces is necessary to be sure that only and exactly the appropriate effects are applied.

[4] Interestingly, this was actually the case for an early incarnation of the `MovingObstacle` representation, but the difficulty of providing a meaningful approximation of acceleration caused it to be stricken from scope early in the development process.

Applicability Via Existing Adaptation Interfaces

As a review, the adaptability interfaces for the `TrafficEstimator` class, represented by the delegate interfaces in Figure 5.6 and by the XPI in Figure 5.7, allow enhancements to be applied to:

1. Whether an obstacle is "in" a given lane of travel, such as overridden by **TE.S.4** to use the `laneAssociations` datum instead of the more generic geometric tests. While `hasV2V` may affect the determination of whether a vehicle is "in" a lane, that effect would likely take the form of an alternate threshold and is addressed by **CX.3** (p.120), rather than an "intrinsic" relevance test.

2. The conservative estimation of an obstacle's travel speed, such as augmented by **TE.S.1** through **TE.S.4**. Again, `hasV2V` will affect this determination, such as through **CX.2** (p.116), but this point of variation is not in itself a "false-positive" culling step.

Moving on to the adaptability interfaces for `PrecedenceEstimator` class, shown in Figures 5.9 and 5.10, supplemental data may affect:

1. Relevance tests for each of the intersection clearance, exit waypoint precedence and cross-traffic yield algorithms. These are definitely relevant to **CX.1**, and the corresponding effects of the `hasV2V` flag will be similar to the "max ignorable speed" effect, **PE.C.1**.

2. Various stages of the somewhat convoluted "obstacle update process", such as used by **PE.S.4** to convey that an intersection is not "quiescent" when it is occupied by observed-moving obstacles. The `hasV2V` datum might make a similar contribution here, but the idea of "quiescence" is more about whether an intersection is "busy" than simply having (possibly stalled) vehicles in it. It follows that something using `turnSignalState`, similar to **CX.5** may be more appropriate in this context. In any case, nothing in the "obstacle update process" constitutes an intrinsic relevance test, so these points of variability are not directly relevant to **CX.1**.

Lastly, the adaptability interfaces in Figures 5.15 and 5.18 for the `MergePlanner` allow augmentation of:

1. The identification of which lane an obstacle is in, analogous to item 1 for the `TrafficEstimator` above and thus irrelevant to **CX.1**;

2. The determination of an obstacle's "culling range", which is also related to item 1 for the `TrafficEstimator`, and is handled by **CX.3**;

3. The augmentation and management of the various intermediate obstacle types, such as was necessary to forward the intermediate "isMoving" state through the processing pipeline for **MP.D.1** through **MP.D.9**. It is possible that these types would have to be similarly augmented with an analogue for `hasV2V`, but only if `hasV2V` affects downstream points of variability in a way that is not already covered by the existing intermediate "isMoving" state, including:

a) The conservative estimate of obstacle velocity, i.e., whether it is permitted to have nonzero velocity;

b) Relatedly, the determination of whether an obstacle is specifically permitted to have "oncoming" velocity;

c) The determination of whether an obstacle requires the minimum vehicle-length "courtesy" gap for a front-merge.

These last three points of variability could all be influenced by `hasV2V`, but they do not amount to "intrinsic false-positive" tests, and are instead covered by other requirements, such as **CX.2** below.

Together, this review of the existing adaptability interfaces yields the somewhat dubious conclusion that, although **CX.1** is a very broadly-scoped requirement, it is only applicable to the `PrecedenceEstimator`, and to comparatively few points of variability therein. This highlights the central issue explored in this chapter by raising the question of whether or not the adaptability interfaces actually expose *all* such relevance tests for augmentation. In the limit, this can only be verified by scrutinizing the source code for the underlying algorithms, which would partially defeat the original purpose of exposing dedicated adaptability interfaces.

For the purposes of this case study, however, a thorough exploration of the underlying implementation is both necessary, in order to apply **CX.1** to the original, "direct encoding" of each component, and also informative, as it suggests several "common" points of variability that could be identified in other algorithms as well. Exposing these in the corresponding adaptability interfaces would enhance their usefulness by reducing the chance that such broad effects as **CX.1** would compel similar expeditions into the implementation of any given core algorithm.

Looking Beyond the Existing Adaptation Interfaces

The core `TrafficEstimator` algorithm is localized in a single method, which iterates over the list of candidate obstacles, tests them for occupancy in Boss's lane of travel, and computes conservative estimates of their rear bumper positions and lane travel speeds for downstream use by the `DistanceKeeper`. The "relevance" of an obstacle is determined entirely according to its identified lane of travel, which is already covered by the existing adaptation interfaces. Thus, no additional work would have to be done in the `TrafficEstimator` to satisfy **CX.1**.

Similarly, the core algorithm for the `PrecedenceEstimator` iterates over the list of candidate obstacles and explicitly tests each one for relevance to the clearance, precedence, and yield contexts before triggering the appropriate calculations. These tests cover all three contexts treated by the `PrecedenceEstimator` that are relevant to moving obstacles, and their representation in the existing adaptation interfaces implies no additional work would be necessary to fulfill **CX.1** through these interfaces.

Lastly, careful analysis of the core `MergePlanner` algorithm does not reveal any "culling" or "relevance" tests beyond the initial "maximum range" determination, as augmented by **MP.S.2** to allow observed-moving obstacles to be included in merge calculations at longer ranges. As this will be covered by **CX.3**, there is also no work to be done for **CX.1** in the `MergePlanner`.

While the critical implication of all of this is that the existing adaptation interfaces are sufficient to express **CX.1**, the above discussion also suggests that many robotic algorithms will have an initial stage of processing that filters candidate inputs against a set of rules for whether or not they are "valid" or "relevant" in their specific context. In order for their adaptability interfaces to be "trustworthy", they should include a way to augment these rules, whether they take the form of explicit validity tests or more implicit measures of "sufficiently close", or both. Even if the base case, embedded in the core algorithm, is simply an unconditional "yes", the effort to expose those tests for adaptability could pay significant dividends when accommodating future supplemental data.

Detailed Implementation Results and Discussion

As the existing adaptation interfaces were sufficient to express **CX.1**, their implementation in the AO and OO designs was a straightforward process. The implementation of this requirement in the DE version was also relatively easy, but only because of the detailed exploration that took place to verify that there were no "latent" relevance tests that were not exposed for adaptation. The raw results of this work are summarized in Table 8.1 in terms of software elements (classes, functions, and lines of code) that were added or modified in each design of the `PrecedenceEstimator`.

Design:	DE	AO	OO
New Lines of Code	2	18	52
Modified Classes	1	1	1
New Classes	0	1	1
Modified Operations	1	1	1
New Operations	0	1	5
Modified Files	1	1	1
New Files	0	1	1

Table 8.1: Raw results from the implementation of Requirement **CX.1**

These results are consistent with the "raw" metric results presented in Chapter 5, in that the AO and OO designs entail significantly more raw source code than the DE approach. They also introduce new classes, and methods therein, where the DE approach remains concentrated in the original `PrecedenceEstimator` class. While this might suggest that the two-line modification of the original, directly-encoded version may be the "best" approach, it is important to remember that those two lines are embedded in a 250-line method of 2000-line class, where the 18 and 52 lines of the AO and OO versions, respectively, are isolated in dedicated modules, making them easier to identify and understand. This is reflected, to a certain extent in the Concern Diffusion results for **CX.1**, shown in Table 8.2.

These are again consistent with previous results, presented in Chapter 6, especially for the "maximum ignorable speed" effect, which, as mentioned above, is very similar to the proposed effect of the `hasV2V` flag. That is, the AO and OO approaches experience slight, but expected increases in diffusion over components and operations, but there are corresponding reductions in diffusion over lines of code that are beneficial to the understandability of the corresponding artifacts.

Design:	DE	OO	AO
Diffusion Over Components (CDC)	1	2	2
Diffusion Over Operations (CDO)	1	2	2
Diffusion Over Lines of Code (CDLOC)	4	3	2

Table 8.2: Concern diffusion results for Requirement **CX.1**

The similarity to the "maximum ignorable speed" effect also means that there is no significant difference in "net option value" for the modules that implement **CX.1** in the AO and OO designs, and that the overall NOV results for each of the three will be largely unaffected. This suggests that the results presented in previous chapters are a reasonably accurate quantification of how well each design can accommodate novel supplemental data, so long as the corresponding effects can be expressed using the existing adaptation interfaces.

If, on the other hand, a proposed supplemental effect cannot be expressed using existing interfaces, there may be wildly varying degrees of refactoring necessary in each *core* algorithm to expose new points of variability. It is not clear that such changes can be easily captured as gradients in the previous metrics, and they would require more significant analysis on par with the contents of the previous three chapters, the repetition of which would be an intractable in the scope of this thesis.

Moreover, the results of such additional analysis would not address the underlying question of whether or not the resulting adaptation interfaces are sufficiently expressive, which was the original purpose of proposing novel input data in this chapter. As such, further detailed implementation and source-level analysis of the remaining supplemental effects in this chapter will be foregone, in favor of focusing on the issue of whether or not the existing adaptation interfaces are sufficiently expressive, and what changes would be necessary if they are not.

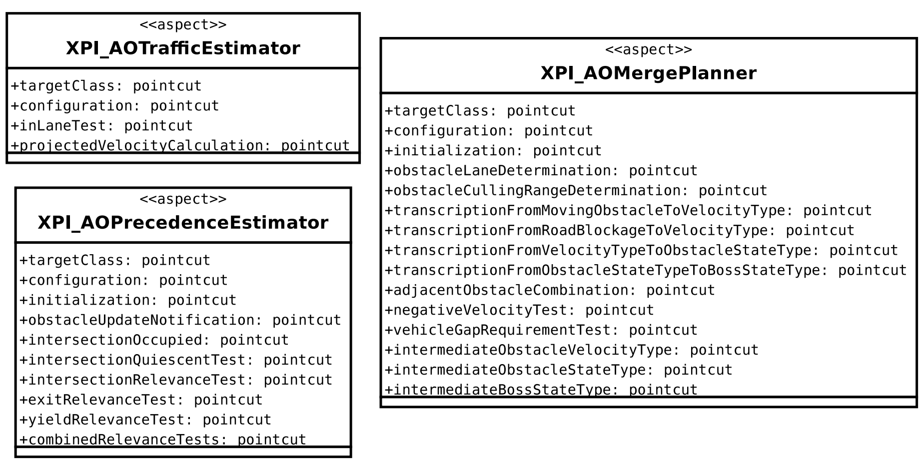


Figure 8.3: Adaptation interfaces for the three behavioral components from Chapter 5.

For reference, the XPI's for each of the three behavioral components refactored in Chapter 5 are reproduced in Figure 8.3. These are analogous to, but more concise than, the corresponding OO delegation interfaces, and will be augmented through the discussion of the six other supplemental effects below. The end result will be a catalogue of "high-value" points of variability in the Urban Challenge software system, which will be distilled into more generic categories at the end of this section. This categorization will guide the identification of similar points of variability in other advanced robotic algorithms in order to ensure that their adaptation interfaces can express a wide variety of future supplemental effects.

HasV2V: Increased Geometric Accuracy

The second candidate supplemental effect evaluated in this chapter is also very broadly scoped, and reflects the same "enhanced" trust that may be placed in the contents of a `MovingObstacle` instance that is backed by V2V, especially in purely proprioceptive data such as wheel speed, curvature, etc.:

CX.2 Context-specific conservative estimation of the position, velocity, and heading of a V2V-supported obstacle shall either be eliminated entirely or restricted to a single standard deviation as reported by `MovingObstacle::poseSTD`, etc., reflecting increased trust in the presence and state of the corresponding vehicle.

Traffic Estimator

The existing adaptation interface for the Traffic Estimator includes the ability to override the estimation of the obstacle's speed, via `projectedVelocityCalculation`, and **CX.2** could easily applied here in the style of existing "moving" and "observed-moving" effects. However, there is no representation of any conservative estimation of the *position* of that obstacle along the lane, even though there is a reasonable expectation that the Traffic Estimator is making such a conservative estimate at some point in its *core* algorithm.

This suggests that the adaptation interface may be incomplete, and close inspection of the core algorithm reveals that the Traffic Estimator does, in fact, include a conservative estimation of the position of the "rear bumper" of a given moving obstacle. This is currently accomplished by projecting the obstacle's center position backwards along Boss's lane of travel by a hard-coded fraction of the estimated "length" of that obstacle. This projection could easily be parameterized to include a degree of "pessimism" to be applied to the obstacle's estimated position, and the determination of that parameter could be exposed for supplemental adaptation according to **CX.2**.

A similar "pessimism" parameter already exists for speed estimation, and should be included in the adaptation interface as well. Interestingly, this parameter is load-time configurable for the Traffic Estimator as "leadVehicleSpeedPessimism_STDs", which suggests that perusal of similar configuration parameters may help identify likely points of variability in other systems.

Rearranging the core algorithm to expose these points of variability would entail a similar amount of work as was done to expose the `projectedVelocityCalculation` pointcut in Figure 8.3, and would yield similar results in terms of raw code size, diffusion, and option value. Thereafter, the extended adaptation interface for the Traffic Estimator would be sufficient to express **CX.2**, along with any other supplemental datum that would affect the degree of "pessimism", as shown in Figure 8.4.

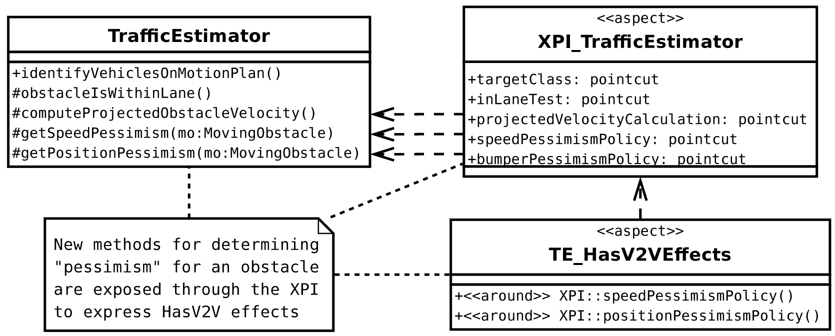


Figure 8.4: Modifications to the Traffic Estimator and its XPI to express **CX.2**

Precedence Estimator

The existing adaptation interface for the Precedence Estimator does not include any way to express candidate effects of **CX.2**, which, from a perspective, is consistent with the fact that obstacles are explicitly tested for relevance in each of the precedence, clearance, and yield contexts before proceeding with the corresponding calculations. Even still, the amount of geometry involved in these calculations suggests that there may be "conservative" estimation of an obstacle's position and/or speed that are not yet exposed for adaptation.

Close inspection reveals that there is, in fact, a single conservative estimation of obstacle velocity that would be affected by **CX.2**. This occurs as part of the calculation of the obstacles "estimated time of arrival", which is compared to the expected transit time through the intersection to determine whether Boss must "yield" to that obstacle. Within that calculation, there is a "conservative" estimate of the obstacle's lane speed that is nearly identical to the speed estimation that occurs in the Traffic Estimator. In fact, it even lends itself to a similar representation as an adaptable "speed pessimism" policy, as shown in 8.5, but the critical difference is that for yield calculations, "pessimism" means "faster", instead of "slower". This highlights the issue of context-specificity of supplemental effects that is one of the principal challenges to robotic software reuse that is addressed by this thesis.

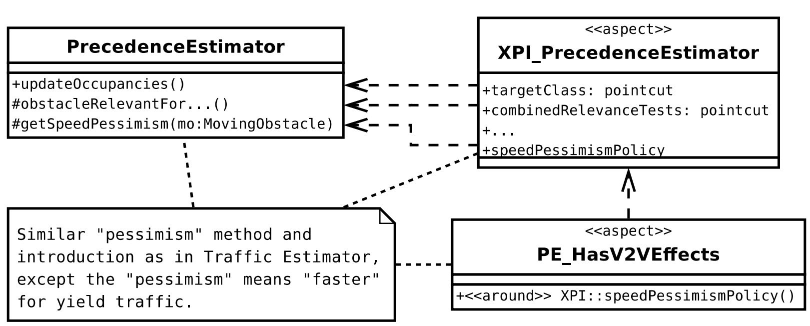


Figure 8.5: Modifications to the Precedence Estimator and its XPI to express **CX.2**

Merge Planner

The Merge Planner's adaptation interface includes the "negative velocity test", which is clearly relevant to **CX.2**, except that this test depends on the contents of an intermediate obstacle type, and not the root `MovingObstacle` representation. This means that `hasV2V` is not readily available at the corresponding stage of processing, and some effort will be required to forward this datum through the Merge Planner's processing pipeline.

An expedient solution might be to simply incorporate `hasV2V` into the determination of the intermediate "isMoving" states, but the semantics of these data are not necessarily compatible with the idea of a V2V-backed obstacle. That is, there are other effects of the intermediate "isMoving" states, such as when converting the contents of a `MovingObstacle` to the intermediate `VelocityType` representation, or for the "courtesy gap" test, that may be adversely affected by simply "blending in" `hasV2V`.

This suggests that `hasV2V` would have to be introduced alongside the intermediate "isMoving" states in order to ensure the semantic integrity of the corresponding supplemental effects. Given the length of the Merge Planner's processing pipeline, as discussed relative to Figure 5.13, this means there is a great deal of "extra" work necessary to propagate the `hasV2V` datum to the point where it can (otherwise easily) affect the "negative velocity test". This would also be necessary for many other supplemental data, such as the `TurnSignalState` datum discussed in Section 8.2, so a more significant evaluation of the Merge Planner's core algorithm is warranted to determine whether any of this "extra" work can be eliminated.

Careful inspection of the Merge Planner's derivation and usage of these intermediate types exposes several issues that will be echoed in the CLARAty case study in Chapter 9, and, in this author's experience, represent recurring themes in advanced robotic software. The resolution of these issues has significant implications beyond the context of the Merge Planner, and a full discussion here would distract from the immediate focus on candidate effects of V2V data. Instead, only the highlights, including the alternate design of the core Merge Planner shown in Figure 8.6, are presented here. The detailed observations and justifications that lead to this design, along with their generalization into guidelines for designing future algorithms to better accommodate supplemental effects, are compiled in Appendix A.

The highlights of this "streamlined" design of the Merge Planner are:

- The intermediate `VelocityType` and `BossStateType` have been eliminated, along with the corresponding need to represent and propagate any supplemental data therein;

- The intermediate `ObstacleStateType` has been renamed to `MergeObstacle`, reflecting the central role that this representation plays in the core merge planning algorithm;

- The "negative velocity test" has been identified as redundant with part of the derivation of the original intermediate `VelocityType`. These have been combined into a more general "conservative" estimation of obstacle velocity, consistent with the Traffic and Precedence Estimators.

- The `MergeObstacle` has been augmented to keep track of the foremost and rearmost `MovingObstacle` instance instead of only the foremost and rearmost "isMoving" state.

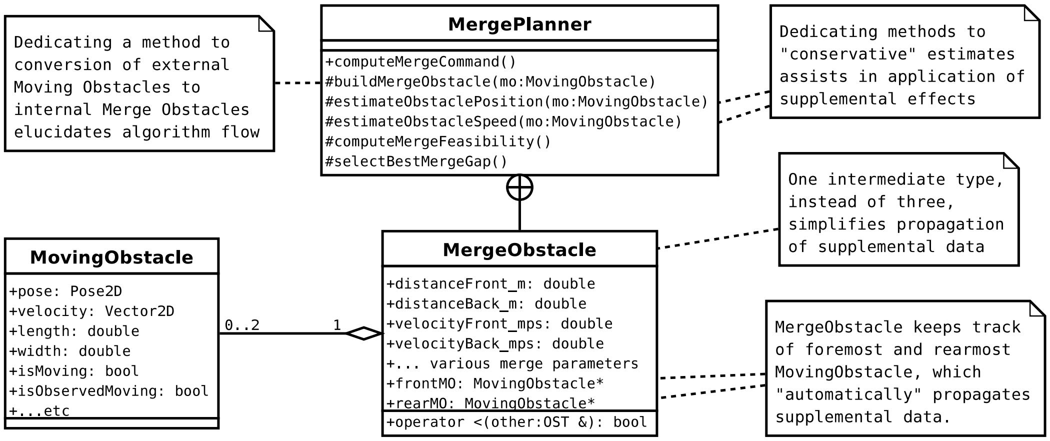


Figure 8.6: Alternate design for the core Merge Planner implementation that simplifies the accommodation of supplemental effects. See Appendix A for details.

The critical benefit of this design, aside from its apparent simplicity, is that all supplemental data in the `MovingObstacle` representation are "automatically" available to downstream policies through the `frontMO` and `rearMO` members of the augmented `MergeObstacle` representation. Thus, *all* of the additional effort put into introducing, deriving, and propagating the intermediate "isMoving" states, or other such supplemental data, is no longer necessary.

This simplifies the the fulfillment of **CX.2** to the level of effort required for the Traffic and Precedence Estimators above. Namely, the estimation of an obstacle's speed and position along a lane would be subject to the same idea of "pessimism" discussed above. The application of **CX.2** would require exposing those values for adaptation, as shown in Figure 8.7.

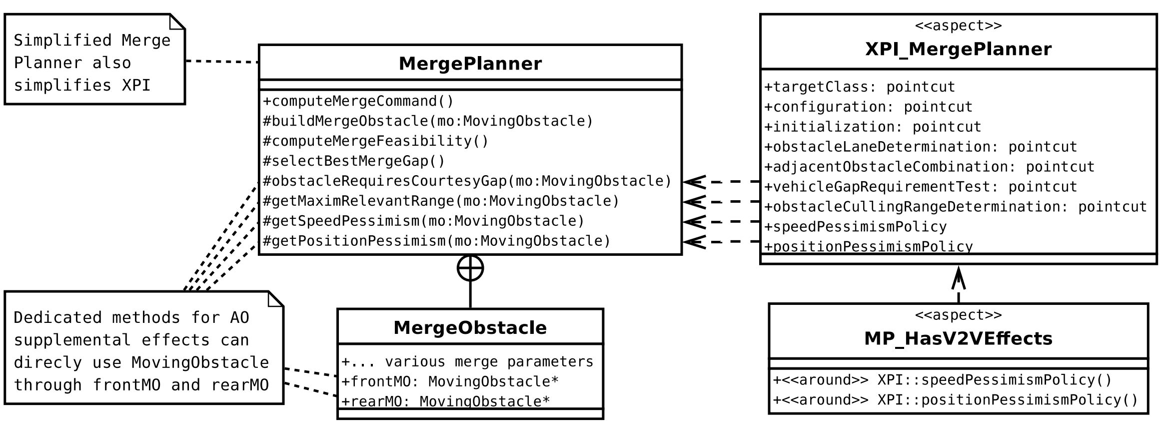


Figure 8.7: Modifications to the Merge Planner and its XPI to express **CX.2**

HasV2V: Relaxed Geometric Constraints

The last broadly-scoped effect of the `hasV2V` flag introduces alternate, configurable thresholds as the third way that the increased confidence associated with V2V signals may impact the inclusion policies for autonomous driving algorithms.

CX.3 Geometric constraints, such as maximum allowable distances, shall be augmented to substitute alternate thresholds for V2V-supported obstacles. These shall be load-time configurable, with default values that relax the associated constraints by 20%, i.e., to increase the chance that a V2V-supported obstacle will be identified as a valid candidate for further computation.

For all three components, the introduction and maintenance of additional configuration parameters is straightforward using both OO and AO designs, such as was done for **TE.S.5**. This reinforces the idea that many supplemental effects may include secondary concerns, such as load time configurability, that should be included in an algorithm's adaptation interface. In fact, the degrees of "pessimism" discussed for **CX.2** could also be framed in this manner, as an alternate and context-specific "gain" on the standard deviation to be used for V2V-backed obstacles. These extra "knobs" enable developers to more finely tune an algorithm as they come to understand the impact of various supplemental data, or to keep up with their semantics as they as they are incrementally *altered* over the course of a project.

Traffic Estimator

As to the specific effects of **CX.3** on the Traffic Estimator, the clearest candidate in the existing XPI is the "in-lane" test, but it is not presently exposed in a way that allows adjustment of any thresholds therein. Close inspection reveals that, at least before the introduction of the `laneAssociations` datum, there were two such thresholds, one on position and one on heading, that would be relevant to this requirement[5]. Exposing these thresholds to apply **CX.3** extends the adaptation interface for the TrafficEstimator as shown in Figure 8.8.

Interestingly, the fact that the `laneAssociations` datum is used to completely override the default geometric tests (per **TE.S.5**) implies that the direct effects of **CX.3** will only be active if the `laneAssociations` datum is removed from the system. This is quite likely as part of future development, as this datum was tied both to the Urban Challenge representation of road lanes, and to the perceptions system's specific models of "nominal" behavior for Urban Challenge Traffic.

Precedence Estimator

As with the application of **CX.2** above, there is nothing in the existing adaptation interfaces for the Precedence Estimator that suggests relevance thresholds that would be affected by **CX.3**. However, given the Precedence Estimator's usage of geometric overlap with "occupancy zones", explained in more detail in [3], there are no explicit "thresholds" to be

[5] In fact, these thresholds were modulated in a manner similar to **CX.3**, using the "observed-moving" property, at a much earlier stage of development.

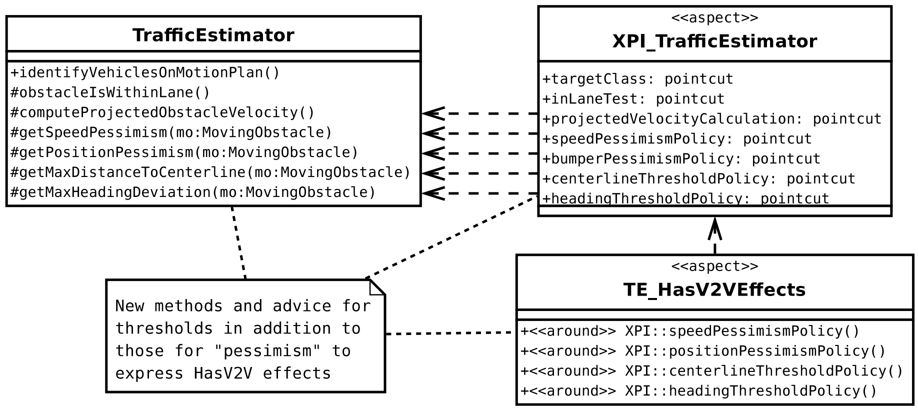


Figure 8.8: Modifications to the Traffic Estimator and its XPI to express **CX.3**

modulated, suggesting that **CX.3** is simply not relevant to the Precedence Estimator. The only way a non-expert could be sure of this, however, would be through careful analysis of the core algorithm, which highlights a different facet of "trustworthiness" of an adaptation interface. That is, in addition to exposing points of variation that *do* exist, there should also be documentation in or near the adaptation interface describing common points of variation, such as inclusion thresholds, that definitely *do not* exist. This would allow future developers to have a certain "confidence" in the coverage of the adaptation interface, allowing them to avoid unnecessary spelunking in the details of the core algorithm.

Merge Planner

For the Merge Planner, the existing adaptation interface already exposes the determination of "culling range" that could be used to express **CX.3** in much the same way as was done for **MP.S.2**. Otherwise, the "in-lane" test includes thresholds that would be modulated by **CX.3**, at least in the absence of the `laneAssociations` datum, in the same manner as discussed for the Traffic Estimator above. Together, these are sufficient to express **CX.3**, and the corresponding extensions to the Merge Planner's XPI, and the application of these effects, are shown in Figure 8.9.

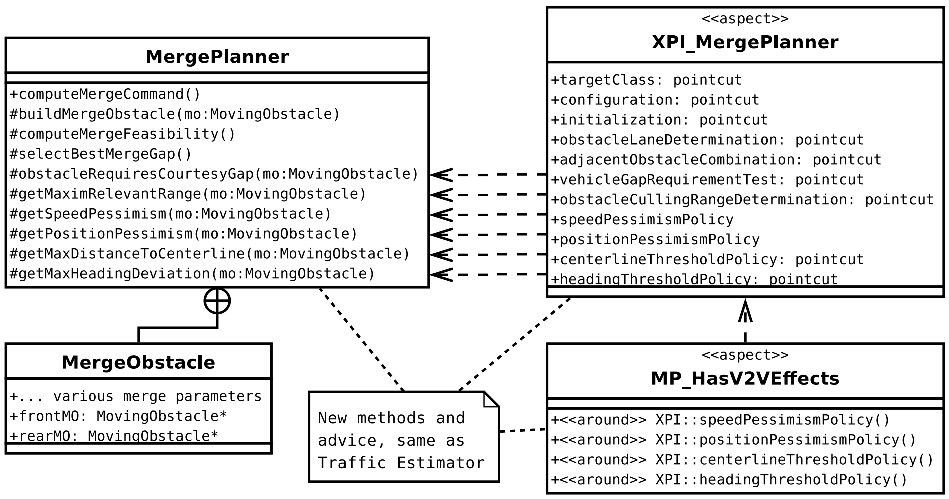


Figure 8.9: Modifications to the Merge Planner and its XPI to express **CX.3**

TurnSignalState: Increased Headway

Where the `hasV2V` flag would have several broadly-scoped effects, the effects of the `TurnSignalState` enumeration discussed in Section 8.2 would be much more algorithm-specific, as they involve reacting to the "perceived intent" of candidate obstacles. The first such effect captures the increased caution that most reasonable drivers practice when a leading vehicle displays nearly any active signal:

CX.4 Obstacles that display active brake lights, turn signals or hazard flashers shall be afforded a larger distance keeping gap to reflect the plausible interruption of smooth forward motion. The degree to which the gap will be increased shall be configurable, with a default value of 20% over the nominal dynamic gap, or headway.

This effect is unique to the distance keeping problem, which is implemented as a collaboration between the Traffic Estimator, which estimates lead vehicle distance and speed, and the Distance Keeper, which uses that information to issue speed regulation commands to the motion planning subsystem. Given this collaboration, there are two distinct ways that **CX.4** may be expressed in this system.

First, turn signal information could be used to affect the "pessimistic" estimation of the lead vehicle's position, such as was done for **CX.2**, but in this case to "imagine" the vehicle's rear bumper to be 20% "closer". This would cause the Distance Keeper to track at a 20% farther distance, which would technically fulfill **CX.4**. The drawback to the expediency of approach is that it slightly *alters* the semantics of the "lead vehicle distance" output, which may have unexpected effects in other consumers of that datum.

The second approach is to directly affect the determination of the appropriate tracking gap for the lead vehicle. While this is obviously the "more correct" solution, this would entail the derivation of a completely new set of adaptation interfaces for the Distance Keeper, which has not yet been addressed in this work. Moreover, this would require the

`TurnSignalState` datum for the "lead vehicle" would have to be forwarded as in intermediate result from the Traffic Estimator so that the Distance Keeper could include turn signal data in its calculations[6].

This resonates with the issues of intermediate data representations discussed for the Merge Planner above. Following that thread, it would be worthwhile to forgo the piecemeal propagation of supplemental data and simply forward the entire `MovingObstacle` instance corresponding to the identified "lead vehicle" as in intermediate result from the Traffic Estimator. This would make supplemental data therein "automatically" available to other observers, including the Distance Keeper, as shown in Figure 8.10:

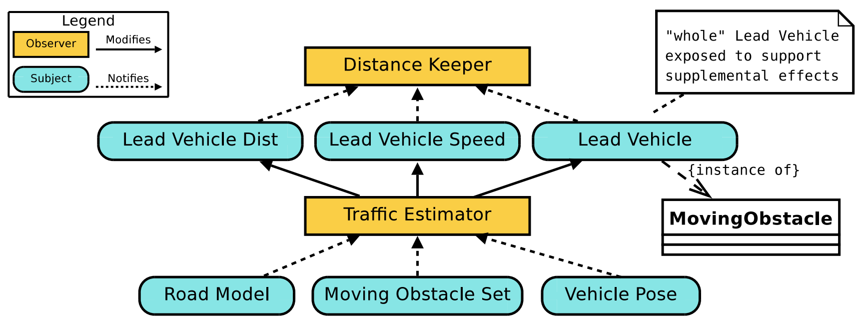


Figure 8.10: Observer collaboration diagram showing augmentation of the Traffic Estimator's outputs to include the "whole" lead vehicle for use by the Distance Keeper.

Given this extension of the Traffic Estimator's outputs, an appropriate adaptation interface may be generated for the Distance Keeper in order to fulfill **CX.4**, such as shown in Figure 8.11.

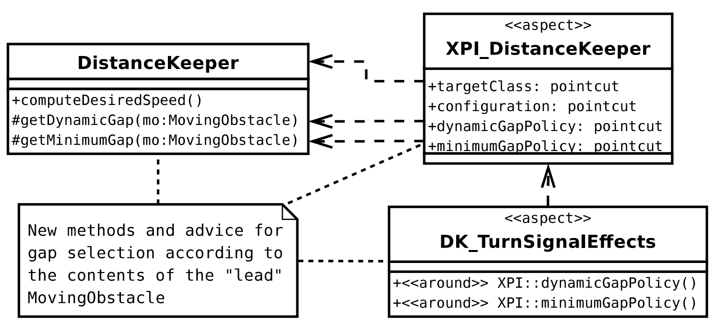


Figure 8.11: Modifications to the Distance Keeper to include an XPI to express **CX.4**

Interestingly, the dynamic tracking gap is already a load-time configurable parameter in the Distance Keeper, so exposing that value as a point of variability would be relatively straightforward, as was the case for the Merge Planners "obstacle culling range", **MP.S.2**. This reinforce the idea that many useful points of variation in an existing algorithm may already be exposed as manually-tunable parameters, and inspecting the Distance Keeper

[6] In fact, there was a short time during active development where the "isPredicted" flag was forwarded to the distance keeper for similar reasons.

for such parameters reveals two configurable "minimum gap" values that specify how far behind a stalled vehicle Boss will come to a stop. These could also be affected by turn-signal or other supplemental data, such as to stop "farther away" from the lead vehicle to allow for easier circumvention maneuvers, so they have been included in the XPI in Figure 8.11 as well.

TurnSignalState: Aggressive Circumvention for Hazard Flashers

Relatedly, the final candidate *additional* effect discussed in this chapter captures the specific semantics of "hazard" flashers to influence the amount of time to wait before initiating a circumvention maneuver around a stopped vehicle:

CX.5 Stopped obstacles that display active hazard flashers shall be circumvented more quickly. This (smaller) alternate circumvention timeout shall be configurable, with a default value of half of the nominal circumvention delay.

While lane-change maneuvers are actually performed by the Merge Planner, the policies for triggering these maneuvers are isolated in a separate component, the Lane Selector. As with **CX.4** above, this would require the generation of an adaptation interface for this class as well, along with somehow propagating the `TurnSignalState` associated with the lead vehicle to the Lane Selector. This further supports the propagation of the "whole" lead vehicle as a separate Subject, as shown in Figure 8.10 above, and doing so would allow a similarly straightforward adaptation interface to be added for the Lane Selector, as shown in Figure 8.12.

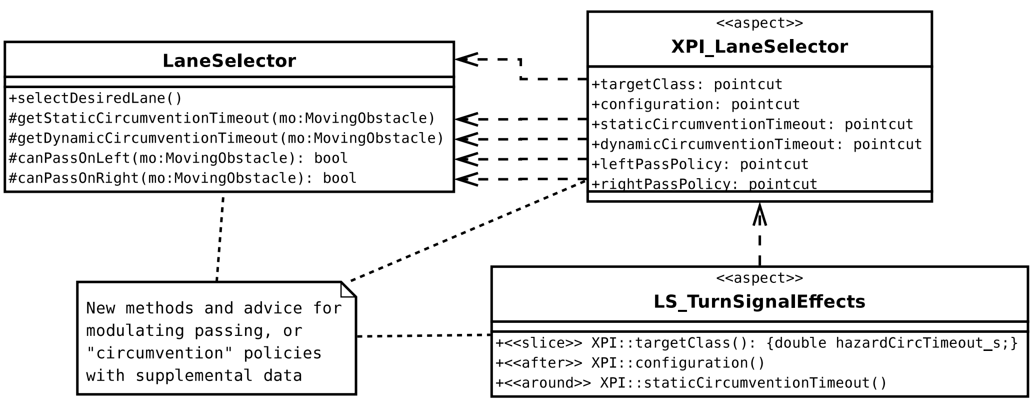


Figure 8.12: Modifications to the Lane Selector to include an XPI to express **CX.5**

Once again, the "circumvention" timeout is already a configurable parameter, and a cursory inspection of other such parameters reveals other candidate points of variation where turn signal and other supplemental data may be applied. These include similar thresholds and timeouts for initiating "dynamic" circumvention maneuvers on multi-lane roads, along with policies for whether or not such maneuvers are allowable, that have also been exposed in the XPI in Figure 8.12. These could be used, for instance, to more readily pass a slow-moving, but not stopped, vehicle with active hazard flashers, or to capture the social convention that it is "rude" to pass a vehicle on the same side as its active turn signal.

8.4 Discussion

The five supplemental effects explored above are not intended to be an exhaustive list of how V2V data might influence autonomous driving behaviors, but rather to provide an array of effects, expressible both within and beyond the original adaptation interfaces from Chapter 5, to begin to piece together some more general guidelines for identifying likely points of variation in robotic algorithms.

First, the more broadly-scoped effects of the proposed `hasV2V` datum suggest three general categories of supplemental effects that pertain to discriminating among a large pool of candidate obstacles, including:

1. False-positive culling or, conversely, context-relevance tests,
2. Context-specific "conservative" estimation of otherwise *primary* data, and
3. Thresholds, timeouts, or "gains" that affect more complicated policies for including, excluding or otherwise incorporating individual candidates in subsequent calculations.

Many of these effects arise from an underlying issue of imperfect perception that is shared by all robotic systems, so it follows that many other advanced robotic algorithms will contain similar policies. Whether for abstract obstacle representations or for more primitive data, such as individual points in a "cloud", this suggests that other algorithms will include context-specific notions of "confidence" that fit one or more of the three categories enumerated above. It is also realistic to assume that many platform-specific capabilities will inform these confidence policies, and that exposing them as part of an algorithm's adaptation interface would allow the modular accommodation of a wide variety of such supplemental data.

For more algorithm-specific effects, such as the modulation of the "circumvention timeout" in **CX.5**, no such categorization is possible, but the discussion above provides two important guidelines for identifying likely points of variability in existing algorithms:

1. Even though they may be highly algorithm-specific, supplemental effects typically modify algorithms in the same way as they do for "confidence" policies, such as augmenting or substituting different thresholds or gains relative to specific properties of a candidate obstacle;
2. At least in existing software, such thresholds are often exposed as load- or run-time configurable parameters, reflecting specific ways that an algorithm might be "influenced" to yield different results.

Together, these provide an excellent starting point for identifying more esoteric points of variability in other algorithms. When merged with the more common "confidence" policies discussed above, and through the consideration of even a few likely candidate effects, a highly expressive adaptation interface may be derived for a wide variety of robotic software components.

In the limit, however, the “best” adaptation interface can only be determined by some combination of prescience and luck that are beyond the reach of even the most experienced designers, and there is always the corresponding risk that the core algorithm will have to be understood and modified to accommodate a future supplemental effect. A minor consequence of this risk is that, in addition to exposing the points of variation that *do* exist, it is important that an adaptation also document common points of variation, such as the “confidence” policies enumerated above, that definitely do *not* exist in a given algorithm. This can save future developers from fruitless exploration of a core algorithm on the chance that such a point of variability *might* be present.

The critical implication of this risk, however, is that the structure of the core algorithm cannot simply be ignored, and is as important a design consideration as the specific contents of the adaptation interface or the detailed techniques for exposing adaptability and binding supplemental effects. Structuring a core algorithm with supplemental effects in mind, such as presented for the Merge Planner in Appendix A, will both simplify the contents of the adaptation interface and can also mitigate the risk of having to bypass that interface by having “prepared” the underlying algorithm for the types of adaptation necessary to accommodate platform-specific details.

This advice for structuring the core algorithm, along with the guidelines for identifying “likely” points of variation, will be revisited in the next chapter, which moves beyond the context of urban driving to evaluate the proposed *primary* vs. *supplemental* methodology as it might be applied to software for planetary (Mars) rovers.

Chapter 9

Complementary Case Study: CLARAty

While the work detailed in previous chapters demonstrates the effectiveness of the proposed *primary* vs. *supplemental* methodology in the context of autonomous urban driving, it is important to consider its applicability to other robotic problem domains as well. In the ideal case, dozens of additional software systems would be subjected to the same redesign, analysis and augmentation experiments described to this point, but the amount of effort required to do so would be intractable in the scope of this thesis. As a more feasible alternative, a small collection of software components from a substantially different robotic problem domain will be analyzed for candidate separations of primary from supplemental data, and the viability of separating the corresponding core algorithms and supplemental effects will be explored at the design level, without delving into the detailed work of refactoring and source analysis as before.

Looking beyond the domain of urban driving, the Coupled Layer Architecture for Robot Autonomy (CLARAty)[38] stands out as one of the most substantial and rigorous attempts to date at providing a wide array of reusable software for a related, but still heterogeneous, class of planetary rovers. Even within this comparatively narrow scope, accommodating the variability in sensors, actuators and mission context has been recognized as one of the key challenges in ongoing development. As such, an investigation of the components within CLARAty should provide a compelling foundation for the general applicability of the methodology proposed by this thesis.

This investigation begins with a review of CLARAty in Section 9.1, including its overall philosophy and design objectives, along with the corresponding design decisions and several important implementation details. Section 9.2 focuses on the modules surrounding the Morphin[59] terrain analysis component, which has already been deployed on multiple systems, and is expected to be deployed on future platforms, that compel the types of adaptation described in Chapter 2. Section 9.3 discusses the existing treatment of primary and supplemental data within the Morphin components and proposes several candidate supplemental effects that may be applied to the algorithm. A rough design consistent with each of the AO and OO approaches described in Chapter 4 is proposed and discussed in Section 9.4, focusing on parallels with the more extensive refactoring done on autonomous driving algorithms in the preceding chapters. Section 9.5 then closes this chapter with more general discussion of the impact that the proposed primary vs. supplemental methodology might have on future work in CLARAty.

9.1 CLARAty: Review

CLARAty was developed with the goal of providing a common infrastructure and an abundance of reusable algorithms that could be shared across various prototype and flight-model Mars rovers. The authors of CLARAty identify the critical challenge in doing so to be the sheer variability of both hardware and software components that comprise robotic systems. Even within the comparatively narrow domain of planetary rovers, they note that neither the intersection of all possible rover capabilities, nor their union, will typically yield a single satisfactory interface[39]. That is, the intersection of all capabilities would leave a component too restricted to use effectively, and the union, even in the unlikely case that it could be determined *a priori*, would yield an intractably large and cumbersome artifact.

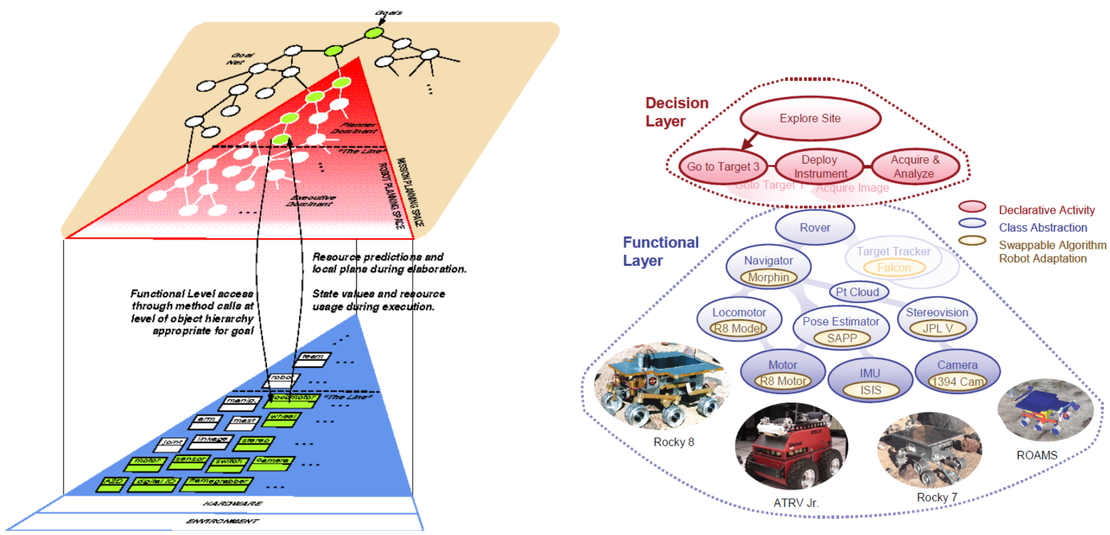
Finding an appropriate balance between these extremes, which is strongly analogous to the discrimination of primary and supplemental data discussed in Chapter 2, was the principal design challenge for CLARAty. As with many other complex systems, the resulting design includes layering and abstractive hierarchies, with the dominant decomposition dividing the system into two distinct layers:

- The *decision layer*, which manages mission goals by planning amongst hierarchical objectives, selecting waypoints for the robot to follow or more fine-grained actions to take, monitoring progress, and reacting to unforeseen circumstances;
- The *functional layer*, which exposes interfaces to the underlying hardware, along with a collection of data processing and navigation algorithms, that are used by the decision layer to manage and execute the mission.

Each of these layers has its own abstractive hierarchy, illustrated in Figure 9.1, and they are "coupled" beyond the typical sense of top- and bottom- interfaces in traditional layered systems[49]. Specifically, the decision layer, while encouraged to use the most abstract functional-layer interfaces possible, is also permitted to "punch through" those functional abstractions as necessary to implement mission- or platform-specific behaviors.

For example, the decision layer typically interfaces to the functional layer at the level of specifying waypoints to the navigation system, but it may also interface directly to the locomotion system to issue lower-level velocity/curvature commands, or even to specific motors to issue fine-grained commands to individual joints or wheels. The trade-off is that the farther down the hierarchy the decision layer goes, the more it must "know" about the underlying platform, and thus the less portable it will be across platforms. In a sense, this is highly analogous to the distinction between primary and supplemental data, except that it is less about the data provided by, and more about actuation capabilities of, the underlying platform.

The functional layer makes extensive use of OO design patterns[21] to provide the decision layer the option of using either abstract, vehicle-independent class interfaces, or else drilling down through the inheritance hierarchy to access a more capable, but also more platform-specific, class interface. Communication between the decision and functional layers is typically achieved via message passing in a cooperating-processes paradigm[49], but



(a) Figure 1 reproduced from [59]

(b) Figure 1 reproduced from [39]

Figure 9.1: The Coupled-Layer Architecture for Robot Autonomy, as depicted in recent publications.

communication within the functional layer is typically via direct method invocation on object instances in a shared memory space. The latter reflects both the need to accommodate the comparatively limited computational resources in a space-worthy planetary rover, and to retain the benefits of strong type-checking in mission-critical software components. Thus, the basic unit of reuse in CLARAty is a single class or a small collection of related classes, which form the reusable "components" that will be the focus of this case study.

The majority of such reusable components reside in the functional layer, with the decision layer viewed as a client of the interfaces exposed therein. Some elements of the decision layer are also reusable, such as the overall framework and planning engine, along with mission-level planning components that make use of the most generic interfaces in the functional layer. The remainder of the decision layer typically uses highly robot-specific interfaces to accomplish highly mission-specific goals (science payloads, etc.), yielding software artifacts that are inherently non-reusable. As with many other facets of software engineering, there is a middle ground between these extremes, especially in the monitoring of platform-specific feedback during the execution of otherwise generic goals. These are typically implemented as individual constraints to planning and execution engines that already exploit known-good domain-specific languages, such as the Task Description Language[50], to effectively encapsulate these concerns.

Parallels between the methodology proposed in this thesis and the various platform-specific constraints and clauses in the decision layer may be worth exploring as part of a future research effort, but for now this work will remain focused on functional layer components, as they are more readily available and more directly focused on reuse between platforms. In particular, the goals of this complementary case study are to:

1. Identify parallels between the OO capability abstractions and the idea of primary and supplemental data (or functionality);

2. Identify any direct treatment of primary vs. supplemental data in existing components;

3. Propose plausible adaptations of the components to alternate data, coupled with discussion of the ways that existing designs would either promote or impede those adaptations.

4. Propose and discuss alternate designs in the style of the refactoring experiments discussed in Chapter 5 that may provide a better treatment of these adaptations.

Two Sides of the Same Coin

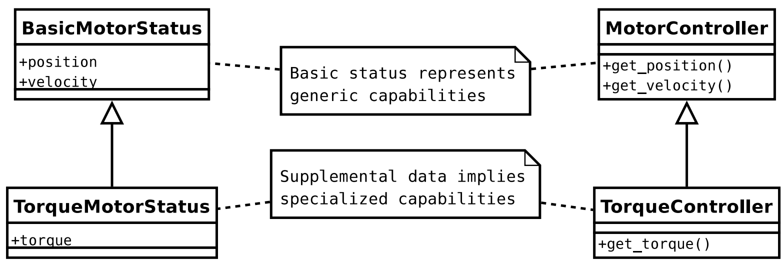


Figure 9.2: The duality of data and capability representations.

There is a very strong duality between the concept of *supplemental* data proposed by this thesis, and the issues of platform-specific capabilities as dealt with in CLARAty. That is, a given supplemental datum can imply the presence of a corresponding specialized capability, and vice-versa, to the extent that they might be seen as data- and function-centric views of the same underlying issue. These even lend themselves to the same kind of hierarchical representation, such as shown in Figure 9.2. Here, the *generic* capabilities and *primary* data each reside in top-level classes, `BasicMotorStatus` and `MotorController`. Specializations of the `MotorController` class can then encode *specific* capabilities, such as the presence of a torque sensor, that are analogous to the `torque` *supplemental* datum in the `TorqueMotorStatus` class.

It is important to note that the specialization of `MotorController` to `TorqueController` on the right of Figure 9.2 is a simplification of the actual patterns used in CLARAty[40], which emphasize the use of the Bridge[21] pattern to allow parallel specialization of the functional interface and the underlying implementation. The critical benefit in this case is that the functional interface may be specialized with logic to control a joint on an arm, such as by incorporating the idea of joint limits, without constraining the interface to the underlying hardware, which may be separately specialized to handle, for instance, a family of related motor controller boards. In either case, the duality of capability and message specification discussed above remains true, reflecting the critical decisions a designer must make when determining the contents of both the generic and specialized classes.

In such cases, both the functionality- and data-centric approaches suffer from similar drawbacks at the extremes of over-generalization and over-simplification. For instance, a functional interface that exposes too many capabilities can become too unwieldy to be practically useful, and may cause client applications to make invalid assumptions about the detailed nature of those capabilities in order to use them effectively. Similarly, a top-level data representation with too many *primary* data may have fields that cannot be populated on some systems, or, perhaps worse, that can only be populated with *similar*, but not *identical*, information that may conflict with assumptions in algorithms that depend on those data, leading to subtly-erroneous results.

There are also similar difficulties at the other end of the spectrum, where an oversimplified functional interface may be too restricted to perform any useful work. This forces client applications to immediately seek more specialized interfaces, defeating the original purpose of declaring the generic one. In parallel, an overly restricted set of *primary* data may force all meaningful functionality into supplemental effects, degrading the associated core algorithm to the point of uselessness.

There is also parallelism in the ambiguities in between, where functional interfaces can emulate one another, such as using a position-based motor controller to emulate velocity-based control and vice-versa, so long as detailed performance and timing characteristics are not critical to the client. This is highly related to the idea of unit- or coordinate-frame conversions discussed in Chapter 2 which, so long as the data are “close enough” to one another, algorithms that consume the data can remain indifferent to the underlying representation.

The key difference between data and capabilities is in the directionality of access and invocation. That is, data are typically *provided to* dependent algorithms, where capabilities are typically *sought out* by client applications. The duality discussed above certainly remains, but this slight difference has critical implications for software adaptation and reuse.

In terms of capability abstractions, a generic capability interface can inherently be reused by all clients that only require generic functionality, and any elements that require more specific capabilities must, in a sense, “know” about those capabilities before seeking them out. As an example from CLARAty, a decision-layer element that performs some robot-specific manipulation task must “already know” about a specific arm or wheel configuration of that robot before seeking it out in the functional layer hierarchy. Thus, there are no “extra” dependencies incurred from the perspective of the functional layer, as they are all collected in the inherently robot- or mission-specific element of the decision layer. In terms of providing a means of incrementally “seeking out” the desired granularity of functionality, the mixture of hierarchical abstraction and polymorphism employed in CLARAty is an excellent approach to the problem of exposing both platform-independent and platform-specific capabilities.

For components that collaborate within the functional layer, such as a stereo vision component that provides obstacle data to a navigation algorithm, CLARAty focuses on a more data-centric approach, emphasizing the use of generic representations of intermediate data products, such as the ubiquitous “point cloud” of obstacle data, to promote interoperability between components. As such, these interfaces are susceptible to the changes in data availability and semantics discussed in Chapter 2, and the consequences of changing the data *provided to* these algorithms less directly addressed by the principles underlying CLARAty.

For example, the principal operating mode of the CLARAty decision layer is to specify a series of waypoints for the robot to traverse as part of achieving some broader science objective. The decision layer monitors the overall progress of the robot, but leaves issues of terrain analysis, path planning and collision avoidance to functional layer components hidden behind a generic `Navigator` interface, such as shown in Figure 9.1b. In contrast to pure capability abstractions such as motor control or locomotion interfaces, this `Navigator` interface composes several advanced robotic algorithms that perform a significant amount of autonomous reasoning, using sensory data, such as from the `Stereovision` component in Figure 9.1b, to detect obstacles and plan and execute safe trajectories. Due to variations in obstacle data sources, such as might be caused by substituting a LADAR point cloud source for the `Stereovision` component in Figure 9.1b, some of these navigation components must be adapted to platform-specific capabilities as they are ported from one robot to the next.

The design of CLARAty accommodates the necessary modifications at a high level through the Bridge[21] pattern discussed above, such as by aggregating a `Morphin` implementation within the `Navigator` component in Figure 9.1b. If faced with an alternate or incompatible "point cloud" representation, a different navigation algorithm could be substituted for the `Morphin` implementation without affecting other components in a CLARAty-based system.

However, the adaptability of such individual algorithm implementations is a lower-level issue that is not directly addressed by the CLARAty design. That is, while the design of the functional layer can certainly accommodate multiple variations of the Morphin algorithm, there are no design guidelines or other mechanisms in place for maximizing the adaptability of the Morphin implementation.

Part of enabling this kind of adaptability is to make use of flexible intermediate data representations, and, to a certain extent, CLARAty uses a variety of template classes that can accommodate many different data types. While some of these will indirectly support the design tasks described in Section 9.4, there is little, if any, usage of templates or other techniques to provide flexibility in input and intermediate data representations for detailed algorithm implementations, which provides fertile ground for applying the methodology proposed in this thesis.

9.2 Morphin Terrain Analysis

Morphin[59] is an algorithm for analyzing the navigability of a patch of terrain given geometric data about the terrain, along with some knowledge of the mobility capability of the robot that is to navigate the terrain. The ultimate output of the Morphin[1] algorithm is a local grid representation of the "goodness" of the surrounding terrain, which is then used as the basis of cost functions for various path planners such as D-Star[53], as summarized in Figure 9.3.

In its most generic form, as is used by CLARAty, Morphin takes geometric terrain data in the form of so-called "point clouds", which are unordered sets of individual three-dimensional (3D) points, each representing a sensor "hit" on a solid object in the real world. These points are passed through several sub-algorithms, such as plane fitting and navigability estimation, to ultimately yield a scalar "goodness" value, where 0 is "bad" and 1 is "good" for a given robot to occupy a given cell.

These point clouds can be the direct output of a sensor, such as a LASER scanner, or else can be derived by processing the outputs of other sensors, such as a stereoscopic camera assembly. As the least-common-denominator of these sensing modalities, the 3D-

[1] Morphin is an enhanced, or "power", version of the "RANGER"[30] algorithm for terrain analysis.

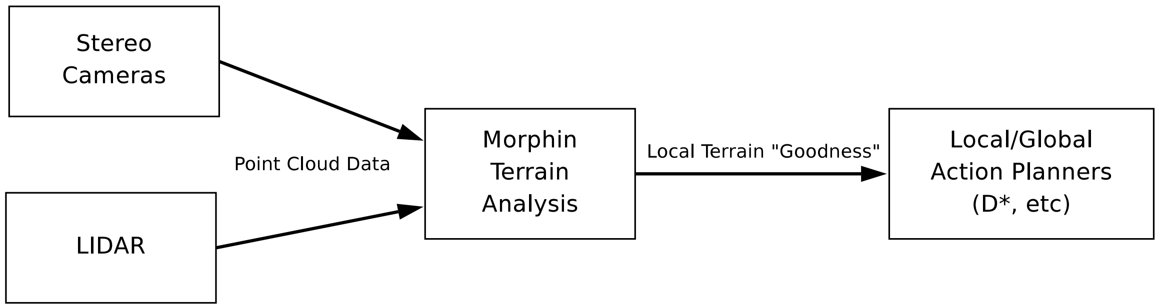


Figure 9.3: Simplified Data-Flow View of the Morphin Algorithm

point representation is sufficiently generic to allow Morphin to be deployed on a wide variety of robotic platforms. For example, Morphin has been deployed on at least three CLARAty robots: Rocky 8, ROAMS, and Fido, by simply converting the outputs of the specific stereo camera assemblies on each robot into the generic point cloud representation expected by the algorithm.

While this speaks strongly to the benefits of generic data representations, the focus on generality comes at the cost of discarding any platform-specific information that may also be relevant to the algorithm. That is, at least conceptually, there is much more about terrain than just its geometry that could influence whether or not it is "good" for traversal by a robot, but *exactly what* will depend on what the robot can detect, and what terrain or terrain features it is expected to encounter. For instance, the existing Morphin implementation also supports the association of error or uncertainty information with incoming points, but this functionality is not connected through the generic CLARAty interface.

As discussed in [39], this is due to the difficulties of correctly converting one type of error measurement to another[2], and of determining a correct "default" value when no error information is available at all. That is, the inconsistency in the presence or type of error information that can be provided by the various laser scanners and stereo camera assemblies on each individual robot makes it difficult to define consistent semantics for an uncertainty value associated with a given point.

From the perspective of the methodology proposed by this thesis, the issues surrounding the treatment of uncertainty information are highly consistent with the issues of adaptation to *additional* data as outlined in Chapter 2. Moreover, the treatment of this uncertainty information is directly encoded in the existing Morphin implementation, and it is conditionally-active, guarded by ad-hoc internal state flags, in a way that is very similar to several supplemental effects identified in autonomous driving software in Chapter 5. That this existing functionality is so readily identifiable indicates that the problems targeted by this thesis, of adaptation to *supplemental* data, are common and under-treated phenomena in advanced robotic software.

Looking beyond this single already-existing example of uncertainty information, it is also easy to imagine other data that a robot might provide that could also influence the

[2] Consider the simple example of position errors in polar coordinates, whose "curved" nature can only be approximated by analogous error values in a Cartesian coordinate frame. While seemingly pedantic, the distinction might be critical to subtle assumptions embedded in an algorithm, and can accordingly lead to subtly-erroneous results.

"goodness" results of the Morphin algorithm, but are not accommodated by either the generic point-cloud interface, or the existing Morphin implementation. For instance, in a stereo vision configuration, all information about luminance, texture, etc., is currently thrown away at the conversion to a point cloud, but these values could conceivably make nontrivial contributions to the "goodness" computation. As a fairly direct example, the underlying cameras may be thermal imagers with the pixel values that correspond to some temperature of the scene. Depending on the mission context, it may be desirable to either seek or avoid various hot or cold spots in the terrain, which can be represented by modulating the "goodness" result of the Morphin algorithm according to an appropriate policy.

Similarly, additional texture analysis or image understanding algorithms might be used that could associate "rockiness", "sandiness", etc. with a given point, which could also be used to influence the "goodness" calculation according to mission context and goals. Looking beyond the scope of Mars rovers, algorithms for vegetation detection, or identification of generic "features of interest", such as trees, people, or automobiles, could also introduce data that can also be influence in the "goodness" calculation. None of these data are supported by the existing CLARAty interface, and all of them would require invasive modification to the Morphin implementation.

When combined with the ad-hoc nature of the existing treatment of uncertainty data, these examples further reinforce the broad applicability of the methodology proposed by this thesis, as no one input specification can possibly represent all data that could possibly be relevant to the Morphin algorithm. It follows that some explicit treatment of adaptability to *supplemental* data is required, and several designs consistent with the methodology proposed in Chapter 2 are presented in Section 9.4. Before looking at these possibilities, however, it is necessary to dig deeper into the details of the existing Morphin implementation in order to understand the stages that the point cloud data go through on their way to the "goodness" result, and how the error information is (and other information may be) incorporated into the corresponding calculations to influence the results.

9.3 Data Flow and Dependencies in Morphin

Expanding on the simplified data flow in Figure 9.3, the processing pipeline in Morphin consists of four critical transformations from raw sensor (stereo camera) data to the ultimate output of a local "goodness" grid. These four steps, plus a fifth stage that converts "goodness" to planner-specific "cost" functions, are summarized in Figure 9.4.

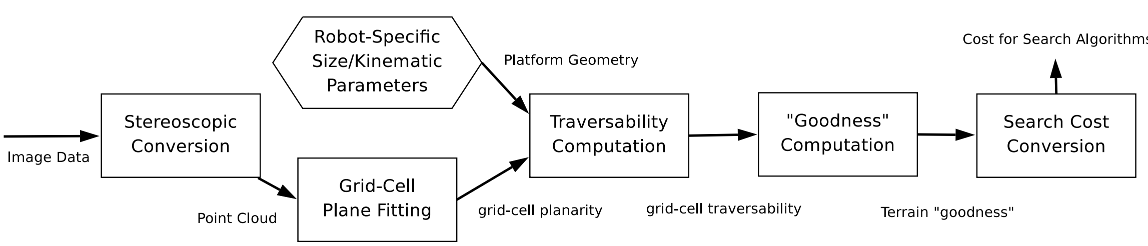


Figure 9.4: Detailed Data-Flow View of the Morphin Algorithm

Each of the conversions presented in Figure 9.4 represents a *semantic interpretation* where some input data are combined with platform details and mission context to yield output data that have new, and typically more abstract, semantics. Consistent with the work on the Merge Planner in Chapter 5, these stages of the core Morphin algorithm can easily be affected by changes in the content or semantics of their input data, and should include some explicit treatment of adaptability according to the methodology proposed in this thesis.

In support of the alternate designs presented in Section 9.4, this section explores the details of the existing Morphin implementation, focusing on the specific artifacts (classes, methods, etc.) that collaborate to perform each stage of interpretation in Figure 9.4. Existing treatment of the supplemental "uncertainty" data discussed above will be highlighted, along with the more general strengths and limitations of the current design, especially at treating the kinds of adaptation discussed in Chapter 2.

Stereoscopic Point Clouds

The first stage of interpretation is the conversion of raw sensor data, in this case stereo camera images, into the generic "point cloud" representation expected by the Morphin implementation. Beyond the `Point` and `Point_Cloud` representations, there are five significant classes involved in this processing stage, and their relationships are presented in Figure 9.5.

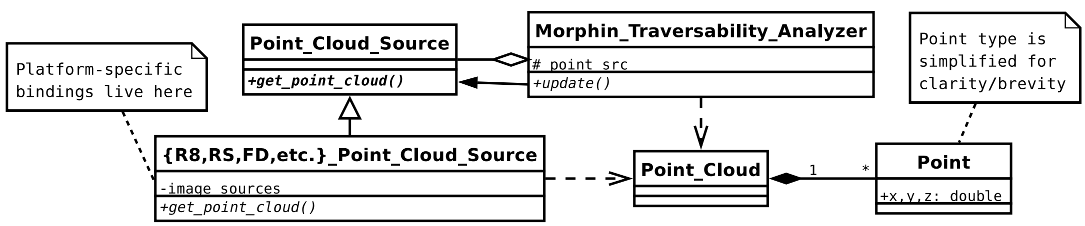


Figure 9.5: Collaboration diagram for conversion of camera images to point clouds for Morphin. Notation: UML

The `Morphin_Traversability_Analyzer` (MTA) is the central element in this and several subsequent conversion stages. That is, the primary purpose of the MTA is to coordinate several subsidiary classes in the Morphin implementation in order to produce new output values when new input data are available. This largely consists of a sequence of method invocations on those subsidiary classes, passing intermediate results from one to another until the processing sequence is complete. There is some deviation from this, such as will be discussed for later processing stages below, but the role of the MTA as a coordination, or Mediator[21], class holds true enough for this discussion.

The MTA aggregates a `Point_Cloud_Source`, which provides a polling interface to get point cloud data from the underlying platform. This interface, *get_point_cloud()* is invoked as part of the *update()* method of the MTA, which passes the resulting `Point_Cloud` to downstream consumers as necessary.

The abstract `Point_Cloud_Source` class is specialized per-platform, where the three platforms implied by {`R8`,`RS`,`FD`} in Figure 9.5 are:

- **R8** The Rocky8 prototype Mars rover;

- **RS** A ROAMS-simulated rover;
- **FD** The Fido prototype Mars rover.

There are several other robots that have similar bindings to the Morphin algorithm, such as discussed in [40], but the source code for these additional platforms is not as readily available as for the three platforms listed above. While access to more examples would certainly be useful, these three are both similar and diverse enough to suffice for this discussion. In terms of their similarities, the overall algorithm in each of the three specializations of `Point_Cloud_Source` is roughly the same:

1. Acquire one or more stereoscopic image pairs;
2. Perform stereo disparity calculation for each image pair;
3. Convert disparity information into the `Point_Cloud` representation;
4. If necessary, fuse two or more `Point_Cloud` instances, one for each stereo pair on the robot, into one coherent point cloud to return as the result of the `get_point_cloud()` method invocation.

The details of each platform are well hidden behind this generic interface, isolating clients from issues such as the intrinsic and extrinsic parameters for each camera, along with the “knowledge” of which cameras comprise a stereo pair. These also allow variations in the exact method of acquiring images, such as interfacing directly to a frame grabber vs. reading intermediate image files from disk, to be hidden from consumers of `Point_Cloud` data. Lastly, these specializations of `Point_Cloud_Source` also encapsulate the configuration and specific usage of different stereo processing algorithms, which must be tuned to available computing resources and camera-specific details of resolution, color depth, expected scene geometry, etc.

In terms of generating purely geometric point-cloud data, this is a highly effective design, as it completely and elegantly encapsulates the derivation of the *primary* geometric data for the Morphin algorithm. The drawback to this design is that there is no means of propagating any *supplemental* data that might yield valuable enhancements to the terrain analysis results. For example, if two thermal imagers were to be used in a stereo arrangement, there would be *additional* data available about the temperature of each point that cannot be represented via the existing `Point_Cloud_Source` interface without forcing all possible platforms to *also* provide some kind of thermal information as well.

More concretely, several of the downstream processing stages, such as the aggregation of `Point_Cloud` data into a plane-fitting representation, include the ability to incorporate uncertainty data, or at least one specific type of “error” information, as mentioned above. However, this functionality is not connected through the `Point_Cloud_Source` interface, largely due to the difficulty of guaranteeing a semantically-consistent estimate of “error” across multiple platforms[39]. While this exclusion allows the Morphin algorithm to be used as-is on many platforms, this comes at the expense of degraded performance, as the incorporation of uncertainty information would yield more accurate results.

Beyond error or thermal information, and especially beyond the context of Mars Rovers, there are many other possible data that could be used to supplement the Morphin algorithm, but are not accommodated by the existing interface, such as:

- Results from simple texture analysis to determine whether some terrain is "rocky" vs. "sandy";

- Results from vegetation detection algorithms, such as [8], or other science instrumentation that suggests terrain that should or should not be driven over.

- Advanced image-understanding algorithms to identify traffic, pedestrians, or other "elements of interest", as discussed above.

Many of these data might be incorporated into the overall navigation algorithm by bypassing the Morphin algorithm entirely and building a parallel "goodness" map that would be fused with the output of Morphin to generate the map ultimately used by the path planning algorithms. While this would certainly be a valid approach in some cases, there are several caveats that must be considered:

1. There would have to be some knowledge shared between the Morphin implementation and this "new" map generator as to the scale and semantics of the "goodness" values such that they may be easily combined (e.g., by a "min" or "max" operator) at the output stage;

2. Much of the innards of the Morphin algorithm, including the generation of the base "goodness" map and the binning of individual points into that map, would have to be replicated in the "new" map generator;

3. A parallel, independent map generator would not have access to the internal details of the Morphin algorithm, and would thus not be able to modulate useful parameters such as "maximum allowable pitch" according to, for example, whether terrain is "sandy" or "rocky".

If any of these caveats are deemed unacceptable, then the only alternative is to extend the `Point` representation and augment the Morphin implementation accordingly. Insofar as the elements in Figure 9.5 are concerned, the critical problem is that, one way or another, some *additional* information would have to be associated with the *primary* geometric data on a per-`Point` basis. This can be accomplished in many different ways, ranging from specialization of the `Point` class to maintaining a separate mapping of `Point` instances to their supplemental data.

However, for the remainder of this discussion, the much simpler (if certainly more invasive!) mechanism of extending the `Point` class to include additional member data will be used, which is consistent with the simplification of the `MovingObstacle` representation in Chapter 5. For example, the existing treatment of uncertainty information in the Morphin algorithm can be accommodated by extending the `Point` class to include an additional double-precision member, simply called "error", which would then be connected to the existing functionality in the next stage of processing. Following the effects of this "error" field through subsequent stages of processing will help to identify critical junctures where other augmentations of the `Point` class might have similar effects.

Grid-Cell Plane Estimation

The next stage of processing in the Morphin algorithm converts the raw `Point_Cloud` data into a grid of ground-plane estimates, including height, pitch, roll and a so-called "residual", which can be thought of as the "lumpiness" of a given cell. As shown in Figure 9.6, the `Morphin_Traversability_Analyzer` (MTA) remains in control of this stage, using the contents of the `Point_Cloud` to populate a `Plane_Fit_Map` through its `add_image_points()` method.

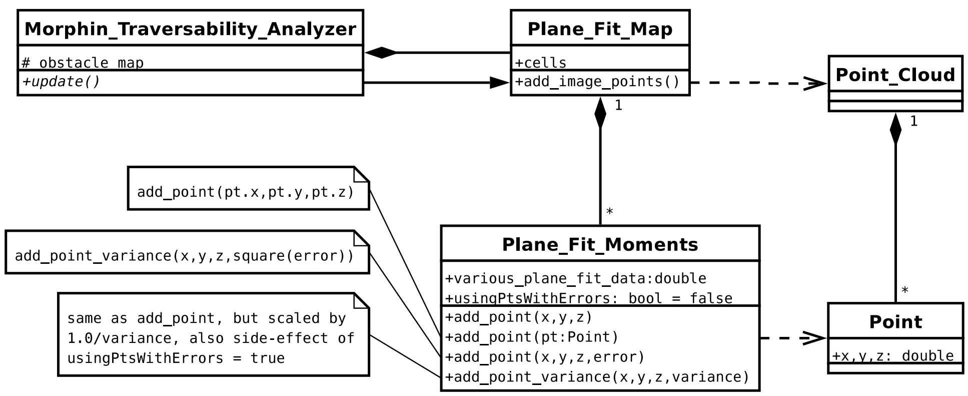


Figure 9.6: Collaboration diagram for conversion of point clouds to plane-fitting estimates for Morphin. Notation: UML

The `add_image_points()` method iterates over the contents of the `Point_Cloud`, using the X and Y coordinates of individual points to bin them into the appropriate grid cells. The points are added to individual cells through the `add_point(pt:Point)` method, which "breaks open" the `Point` representation and forwards its contents to the more verbose variant: `add_point(x,y,z:double)`. This method performs the actual work of adding a single point to the intermediate "plane fitting moments" that are later used to extract the fitted plane, using a derivative of the classic plane-fitting algorithm in [45].

While this level of indirection is somewhat unnecessary, it is a fairly common way of accommodating multiple variations of otherwise semantically-identical data. In addition to "breaking open" an external data type, this type of indirection is also used to implement simple unit- or coordinate-frame transformations such as discussed relative to *primary* data in Chapter 2.

Beyond this treatment of the *primary* geometric data, there are two variations of `add_point` in the `Plane_Fit_Moments` class that are not used by the CLARAty implementation of Morphin. These variations represent an optional, or *supplemental*, incorporation of error information into the plane estimation algorithm:

1. The overloaded method,`add_point(x,y,z,error:double)`, which squares "error" and forwards to:

2. The dedicated method, `add_point_variance(x,y,z,variance:double)`

The differences between these and the geometry-only versions of `add_point` are small, but significant. These can be summarized as:

1. Scale the point's contribution to the plane fitting data by 1.0/*variance*, thereby assigning higher weight to points with lower variance.

2. Set the member variable `usingPtsWithErrors` to *true*, which triggers downstream functionality discussed below.

These are highly consistent with the supplemental effects identified for urban driving software in Chapter 5, which underscores a broader applicability for the *primary* vs. *supplemental* methodology proposed by this thesis. Moreover, this is typical of how supplemental data are often treated in existing systems: by introducing variations of a baseline method and setting additional triggers for downstream processing.

The trouble with this approach is that, while effective for a single or small collection of supplemental data, this can lead quickly to a combinatoric explosion of input methods and possibly-active downstream effects that is simply impossible to manage. For example, adding thermal information, as discussed above, would require:

1. Some cached indication, analogous to `usingPtsWithErrors` in Figure 9.6, of whether or not temperature information has been injected;

2. An extension of the `Plane_Fit_Moments` class to include some aggregation of temperature data, such as average and/or maximum temperatures of the points that contribute to a plane, as temperature is likely to be relevant to downstream computation than to the geometry of plane fitting.

3. A new method on the order of `add_point_temperature(x,y,z,temp:double)`, for points with just temperature information, *and*

4. A method such as `add_point_temperature_variance(x,y,z,temp,variance)`, for points that have both temperature and error information.

The methodology proposed in this thesis would, instead of this ad hoc treatment, identify the incorporation of individual points into the plane-fitting representation as a critical point of variability in the Morphin implementation, to be exposed as part of the Morphin "adaptability interface", such as presented in Section 9.4 . The policies for incorporating error, temperature, or other platform-specific *supplemental* data into the `Plane_Fit_Moments` representation could then be encapsulated in separate classes or aspects according to the design patterns presented in Chapter 4. This would allow results from those supplemental data to be propagated to the next stage of processing without introducing any direct dependencies in the *core* Morphin implementation, which would remain reusable on any platform that can populate the *primary* Cartesian `Point` representation. This also enhances the understandability of the core Morphin implementation by freeing it from extraneous, inactive, or otherwise confusing functionality, such as the `add_point_variance()` method in Figure 9.6, which obfuscates the process of incorporating a `Point` into a `Plane`.

Traversability Estimation

`Morphin_Traversability_Analyzer` continues its controlling role by propagating the plane-fitting results to the `Morphin` class, which, despite its simple and seemingly all-encompassing name, is only one among many subsidiary classes in the overall Morphin implementation. This class uses the plane-fitting representation to derive a set of "traversability" parameters, which are then forwarded to the ultimate "goodness" calculation. The classes that participate in the estimation of a grid cell's "traversability" are shown in Figure 9.7.

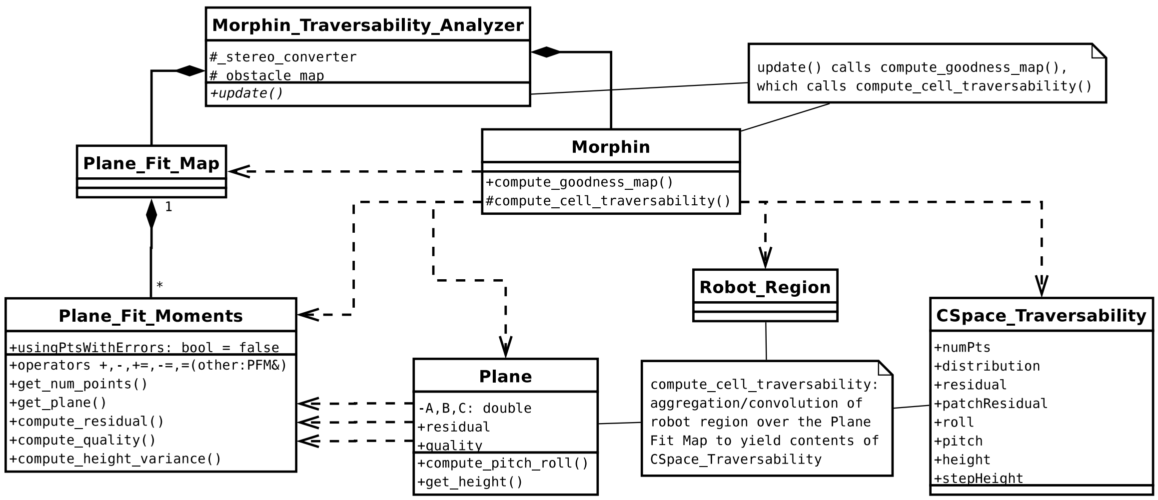


Figure 9.7: Collaboration diagram for conversion of plane-fitting estimates to a local "traversability" representation for Morphin. Notation: UML

The estimation of the traversability occurs within the `compute_goodness_map()` method of the `Morphin` class, but is analyzed separately from the actual "goodness" calculation because of the use of an intermediate type, `CSpace_Traversability`. The ultimate "goodness" results depend exclusively on the contents of this intermediate type, as the algorithm in `compute_goodness_map()` can be summarized as, for each cell in the grid:

1. Perform a configuration-space expansion by aggregating all plane-fitting cells within the robot's "region", or bounding box, into a single macro-plane.

2. Cache the properties of that macro-plane in a `CSpace_Traversability` instance.

3. Compute the output "goodness" cell using the contents of that `CSpace_Traversability` instance, which is discarded before proceeding to the next cell.

The contents of this intermediate data type mostly pertain to the geometric properties of the macro-plane discussed above, including the estimated roll, pitch, and height, as made directly available through the `Plane` representation. Beyond these basic geometric properties, the `CSpace_Traversability` representation also includes the *residual* of the plane-fitting process, which is the $\frac{\chi^2}{N}$ error measurement, roughly analogous to the "lumpiness" of the aggregated plane. There is also a representation of how well the plane is covered by sensor data, both in terms of the number of points that contribute to the estimated ground plane, along with how well they are distributed over the plane, as opposed to being clustered in a corner and leaving the rest ill-defined. Lastly, the `CSpace_Traversability` class captures the worst-case "residual" of the individual sub-planes, along with the height difference between the highest and lowest sub-planes.

This representation provides reasonable coverage of what would make a region of terrain "good" or "bad" for traversal. Each of the properties of the `CSpace_Traversability` representation would be subtly affected by adding error information to the plane estimation process, but those effects would be transparent to the goodness calculation. That is, the effects of introducing error information into the `Point` representation can be thought of as being "contained within" or "hidden behind" behind the contents of the `CSpace_Traversability` without "losing" any of the semantics of the "error" data.

The exception to this is that the *quality* member of the `Plane` representation is absent from all calculations that contribute to members of the `CSpace_Traversability` class. Close inspection reveals this member to be computed only if `usingPtsWithErrors` has been set, so the introduction of this *quality* member might be viewed as a *supplemental effect* that is very similar to the introduction of the intermediate "isMoving" states for the Merge Planner (see, e.g., **MP.D.1**).

In terms of semantics, "quality" in this case seems to be a complex function that models how well the "lumpiness" of the fitted plane can be "explained" by the error estimates provided with the point data. Even though it is not currently incorporated into the calculation of the `CSpace_Traversability` contents, such information could easily be relevant to the subsequent "goodness" calculation, perhaps triggering more conservative estimates thereof in the case of particularly ambiguous fits.

Still, this is at least partially redundant with the existing "residual" information, which, when combined with the lack of any other means of propagating the "quality" result through the `CSpace_Traversability` representation, seems a likely explanation of its exclusion from further calculations. As with the population of the `Plane_Fit_Moments` class above, the methodology proposed in this thesis would identify the augmentation and population of the `CSpace_Traversability` representation as critical points of variability in the Morphin implementation. Including these in the AO or OO adaptability interfaces would allow the effects of error, thermal, or other supplemental data to be propagated to the final stage: the "goodness" calculation.

Goodness Estimation

As mentioned above, the "goodness" calculation follows immediately after the population of the `CSpace_Traversability` class in the `compute_goodness_map()` method of the `Morphin` class. As such, the collaboration diagram in Figure 9.8 only introduces one new class, the `Goodness_Cell` representation, which is used to hold the final results.

This is the final stage of the Morphin algorithm, where the highly abstract concept of "goodness" is extracted from the largely geometric information about roll, pitch, height, and the "residual", or lumpiness, of the terrain. Interestingly, "goodness" is actually calculated as the complement of the "badness" of a given cell, which is the worst case of three different "hazard" calculations, which are restricted to the range $[0, 1]$, corresponding to a smooth range from "safe" to "hazardous".

The first two hazards are straightforward "roll" and "pitch" hazards, wherein the corresponding fields of the `CSpace_Traversability` are scaled according to some minimum and maximum values. Angles under the minimum are assigned a hazard of 0, and angles above the maximum are assigned a hazard of 1, with angles between linearly interpolated. The min and max values are configurable, and are presumably tuned to the specific characteristics

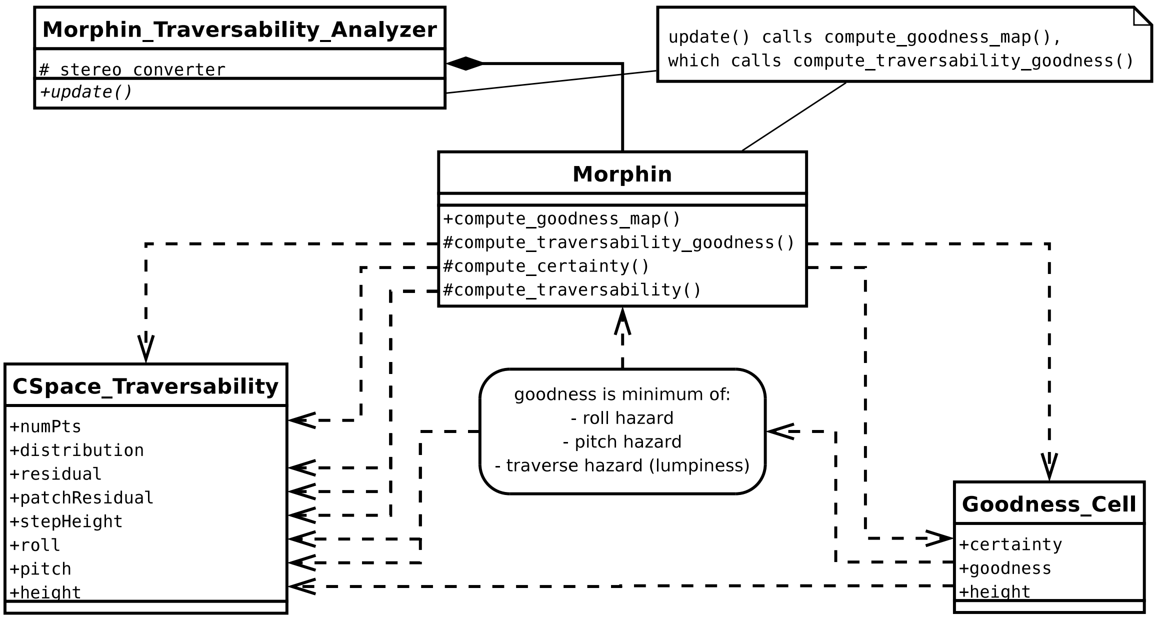


Figure 9.8: Collaboration diagram for conversion of local "traversability" to "goodness" for Morphin. Notation: UML

of the robot and expected terrain.

The third hazard that contributes to the goodness estimate, the so-called "traverse hazard", is a similar function of the members of `CSpace_Traversability` that encode "lumpiness": the residual, patchResidual, and stepHeight members. These are similarly scaled against configurable thresholds into the range of $[0, 1]$, and the overall "goodness" result computed as $(1 - max(roll\_hazard, pitch\_hazard, traverse\_hazard))$.

The thresholds used for calculating the individual hazards are very similar to the configurable thresholds for relevance tests in the urban driving software discussed in Chapter 5, and this is one significant way that supplemental data may be used to affect the overall results. For example, the identification of "sandy" terrain may be used to adjust the baseline thresholds, or simply substitute alternate thresholds for the maximum acceptable pitch or roll angles, according to an increased risk of sliding off course as opposed to more firm terrain.

Relatedly, this would be an effective place to apply the "quality" measure that is available after the plane-fitting stage when incoming points are annotated with error information. If this "quality" measure were propagated through the `CSpace_Traversability` representation, it could be used to bias the traverse hazard, or even the simpler roll and pitch hazards, to represent an increased risk in an areas of low quality or high uncertainty. Similarly, the individual hazards estimates might be adjusted to avoid areas with "a lot" of vegetation, either because the vegetation adds some uncertainty as to the "true" nature of the terrain, or, from a more scientific standpoint, it may be undesirable to drive over vegetation that may be investigated at a later time.

Beyond the modulation of individual hazards, the "worst case hazard" reasoning framework could be extended to allow for completely new hazards derived from *supplemental* data, such as adding a "melt hazard", which would be relevant for a volcano-exploring robot, wherein thermal information, once propagated and/or aggregated through the three preceding stages described in this section, could be used at this final stage to convey "this terrain is too hot to risk driving over".

In addition to the principal "goodness" output, there are also "certainty" and "height" members of the `Goodness_Cell` representation that may be affected by supplemental data as well. The "certainty" member is currently a strict function of the number of points that contribute to the cell, but could also conceivably be affected by, "quality", "vegetation" or other supplemental data described above. The "height" member is simply forwarded from the homonymous member of the `CSpace_Traversability` class. Cursory investigation shows that this member is used in visualizations of the Morphin results, suggesting that regarding temperature, vegetation, etc., might also be worth introducing into the `Goodness_Cell` representation for similar purposes.

In the current design, however, there is no explicit accommodation of supplemental data or effects, and each of the above modifications can only be accomplished by invasive modification of the `Morphin` class. In contrast, the methodology proposed by this thesis would instead expose an adaptability interface that includes the individual hazard calculations, along with their aggregation according to the "worst case" rule, as high-value points of variability that could be influenced by *supplemental* data. The various effects described above could then be encapsulated in separate classes or aspects, allowing the supplemental data to influence the final "goodness" results without introducing direct dependencies at any stage of the *core* Morphin algorithm.

Cost Conversion

While the `Goodness_Cell` representation can be thought of as the ultimate output of the Morphin algorithm, there remains one additional step to convert this representation into something usable by the search, planning, and visualization algorithms that consume it. To understand how the goodness results are propagated to these other components, it is necessary to broaden the scope of the collaboration diagrams to include the top-level `Navigator` class, which coordinates the entire waypoint-based navigation process.

As shown in Figure 9.9, the `Navigator` class aggregates a `Traversability_Analyzer` and an `Action_Selector`, and manages their interactions in a sense-plan-act cycle as part of the `nav_loop()` method. The actual connection between Morphin and the action selection algorithm occurs through dedicated "function objects", as they are called in [59], which provide the `Action_Selector` with abstract interfaces for querying the expected costs of candidate local and global actions.

These function objects are very similar to the OO design for encapsulating supplemental effects presented in Chapter 4, in that the derivation of cost is delegated according to the Template Method and Strategy patterns[21] to individual classes such as `DStar_Global_Cost`. These classes are dedicated to the task of converting some underlying representation, such as "goodness", traversability, etc., into the concept of "cost" expected by the consuming algorithm.

While these comparatively small classes can be easily adapted to changes in content or semantics of the underlying representation, the encapsulation of cost-calculation policies is not complete. For instance, there is a cost calculation embedded in the `Goodness_Map` repre-

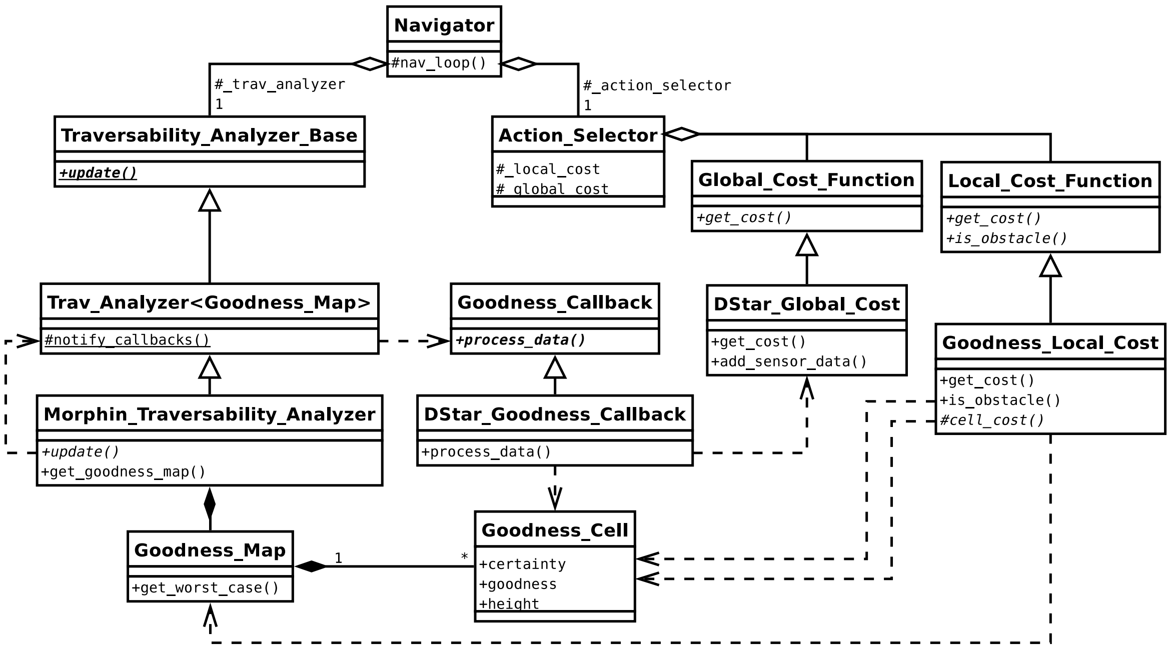


Figure 9.9: Collaboration diagram for propagation of Morphin "goodness" to consuming planners via cost functors. Notation: UML

sentation, as part of its `get_worst_case()` method, that is redundant with the functionality embedded in the `DStar_Global_Cost` functor, reflecting a less-than-ideal separation of the D-Star cost "concern". Close inspection of the implementation, however, indicates that this may have been the original intent, but the typical constraints of time and budget may have prevented it from achieving parity with the ideal design.

Nevertheless, similar less-than-ideal divisions of responsibility are present among many of the classes discussed in this section, ranging from stereo conversion all the way to the traversability representation, that would make adaptation even more difficult than otherwise implied in this discussion. In this particular case, it is unlikely that a supplemental effect would have to be propagated through the `Goodness_Cell` representation, as the "goodness" representation is both simple and powerful enough to accommodate the vast majority of terrain features that a robot might be able to detect or discern.

Still, as was the case with the urban driving software discussed in Chapter 5, a certain amount of refactoring would be necessary in order to allow a clean, concise exposure of adaptability in the Morphin sub-algorithms. These issues are highlighted as part of the high-level discussion in the next section, which applies the AO and OO design techniques proposed in Chapter 4 to the components discussed above, drawing parallels to the more detailed work done on urban driving software in order to highlight the relative strengths of each approach and the applicability of the overall methodology proposed by this thesis.

9.4 Alternate Designs for Morphin

For the urban driving software that was refactored in Chapter 5, the existing structure was preserved to the greatest extent possible in order to avoid any sense of "gaming" the metrics presented through Chapter 7. However, a significant result of these analyses, and of the candidate augmentations to include V2V data discussed in Chapter 8, is that the difficulty of accommodating supplemental data and effects depends a great deal on the detailed structure of the core algorithm.

In applying these techniques to other software components, such as the Morphin implementation discussed above, there is no reason to be so tightly constrained by the existing structure. In fact, some designs that would otherwise be considered "good practice", such as the Merge Planner's insulative use of intermediate data types, discussed in Section 5.4, can adversely affect the adaptability of an algorithm to supplemental data. It follows that an initial refactoring, independent of any particular approach for encapsulating supplemental effects, would be a valuable first step to enhancing the adaptability of the Morphin algorithm. This process should focus on:

- Removing any pre-existing supplemental effects (and setting them aside for later re-integration),
- Reducing or eliminating the usage of intermediate data types that would have to be augmented to accommodate supplemental data,
- Ensuring a good (one to one) mapping between individual processing stages and the methods that perform them, and
- Streamlining the path of input data into the system, particularly avoiding unnecessary levels of indirection or prematurely "breaking out" the contents of an input representation, such as the `MovingObstacle` or `Point` classes, that may later be augmented to include supplemental data.

Figure 9.10 shows one of the many ways these guidelines may be applied to Morphin design, wherein critical changes for each class are:

- The `Morphin_Traversability_Analyzer` class (`MTA`) no longer owns or manipulates a `Plane_Fit_Map`. The process of binning points from a cloud into the plane-fitting representation is now hidden behind the `Morphin` class. This has the joint benefits of consolidating the core algorithm in the `Morphin` class and leaving the `MTA` class much more narrowly focused on the job of binding the Morphin algorithm to the role of "traversability analyzer" in CLARAty. Relatedly, the instance of the `Morphin` class aggregated by the `MTA` has been renamed to `_morphin_impl` to more accurately reflect that the *actual* implementation of the Morphin algorithm is embodied therein.
- The `Morphin` class absorbs the responsibility of binning points into plane-fitting cells, exposing the corresponding `add_points` method as the one and only conduit for input data. The homonymous method in the original `Plane_Fit_Map` class has been stripped out, relieving that class of an explicit dependency on the `Point_Cloud` representation. Instead, the `add_points` method of the `Morphin` class iterates over the individual

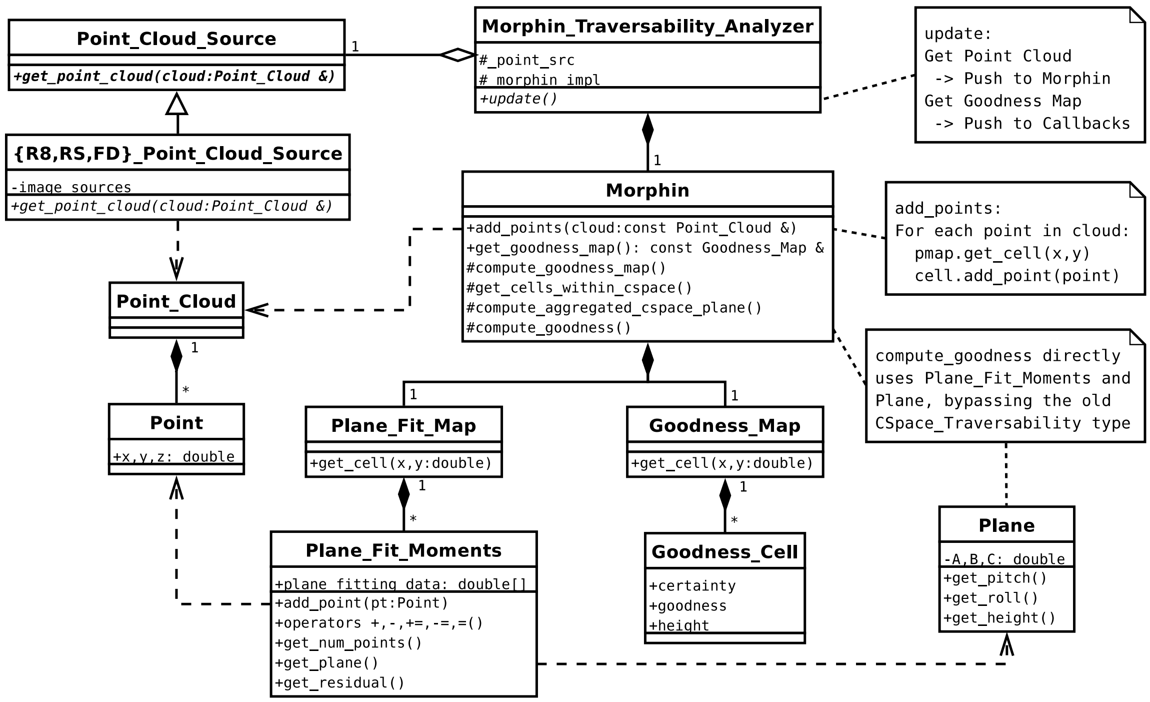


Figure 9.10: Suggested redesign of the Morphin algorithm as a precursor to the application of advanced approaches for encapsulating supplemental effects.

points in the cloud, using the X and Y coordinates of each point to look up the appropriate plane-fitting cell, and adding that point to that cell. Thereafter, it invokes the `compute_goodness_map` method, which in turn invokes the series of subsidiary methods that have been introduced for each stage of processing discussed above.

- The `Plane_Fit_Map` class, in losing its `add_points` method, is reduced to a generic grid map of `Plane_Fit_Moments` instances. This grid-map functionality is derived from an external `Grid_Map` template, also used by the `Goodness_Map` class, which is omitted from Figure 9.10 for clarity. Relatedly, the `get_worst_case()` method of the `Goodness_Map` class has also been removed, as it was only used by, and thus be cleanly transplanted to, the `Goodness_Local_Cost` class from Figure 9.9. This relieves the `Goodness_Map` class of any "cost conversion" policy and aligns the design more cleanly with the apparent intent of the cost functors.

- The `Plane_Fit_Moments` class has been stripped of its `usingPtsWithErrors` member, along with the "error" and "variance" derivatives of `add_point`, as part of removing the supplemental effects of "error" data discussed above. Relatedly, the `compute_quality` and `compute_height_variance` methods have also been removed, as they were triggered by the `usingPtsWithErrors` member. The explicit overload, `add_point(x,y,z:double)`, has also been removed, focusing the plane fitting representation on a single `add_point` method that depends on the external `Point` data type. This streamlines the path that the `Point` representation takes on its way into the Morphin algorithm, ensuring that any supplemental data introduced into the `Point` representation will be available at this stage of processing.

- Lastly, the `quality` member of the `Plane` class has been omitted, as it was populated by the `compute_quality` member of `Plane_Fit_Moments`, and the intermediate `CSpace_Traversability` representation has been eliminated, as it was only used ephemerally to collate results that are now directly available through accessors in the `Plane` and/or `Plane_Fit_Moments` classes.

This alternate design preserves both the overall role and the detailed functionality expected by other CLARAty components. That is, the effects of "error" information were not active in the original implementation, so their excision does not affect the output "goodness" results. The rest of the changes are hidden behind the `Morphin_Traversability_Analyzer` interface, so this alternate design could be implemented and substituted for the current Morphin implementation in CLARAty without any ramifications to other software components.

Even without the application of more advanced design techniques, such as the AO and OO designs presented in Chapter 4, the enhanced coherency of this design would make it more adaptable to variation in the input `Point` representation than the original design for the Morphin algorithm. For example, the collection of all Morphin "policy" within the `Morphin` class, and the subdivision of critical processing stages into separate methods therein, does a better job of highlighting the trajectory of `Point` data through the algorithm than the previous design, which sprinkled those policies across several more generic methods of several more disparate classes. This makes it easier to locate and understand places where supplemental effects might be introduced, updated or removed, regardless of the detailed approach to their implementation.

With this alternate design as a fresh starting point, it is now possible to start discussing AO and OO adaptability interfaces. Before doing so, however, it is necessary to address the means by which alternate `Point` representations might be maintained and introduced for each of the individual robots supported by CLARAty.

Flexible Input Representation

The loosely-coupled nature of the urban driving software discussed in previous chapters allowed the details of having a flexible `MovingObstacle` representation to be largely ignored as an "external" concern. That is, the issues of introducing and populating supplemental data in the `MovingObstacle` class were taken for granted in favor of focusing on their effects on the software components under investigation.

In contrast, the Morphin implementation is much more tightly-coupled, partially due to the overall focus on class-level component integration in CLARAty, but also because the Morphin algorithm is particularly data-intensive. That is, the Morphin algorithm may be required to efficiently process many thousands of points in a single "cloud", where the urban driving algorithms were not expected to deal with more than 50 moving obstacles in any given situation. This precludes the use of a polymorphic `Point` representation, as the associated increases in memory and access time would yield unacceptable overall performance. It follows that the flexible accommodation of alternate input data types is a much more important design consideration for Morphin, which cannot be simply ignored as before.

In support of this tighter coupling, CLARAty already makes significant use of templates and other generic programming techniques to allow high-level classes, such as the `Navigator` class in Figure 9.9, to be easily composed of platform-specific implementations of generic functional interfaces, such as the `Morphin` specialization of the `Traversability_Analyzer` interface. Following this trend, it is possible to adjust the design presented in Figure 9.10 to align the input specification behind a more generic (templated) representation of a "point".

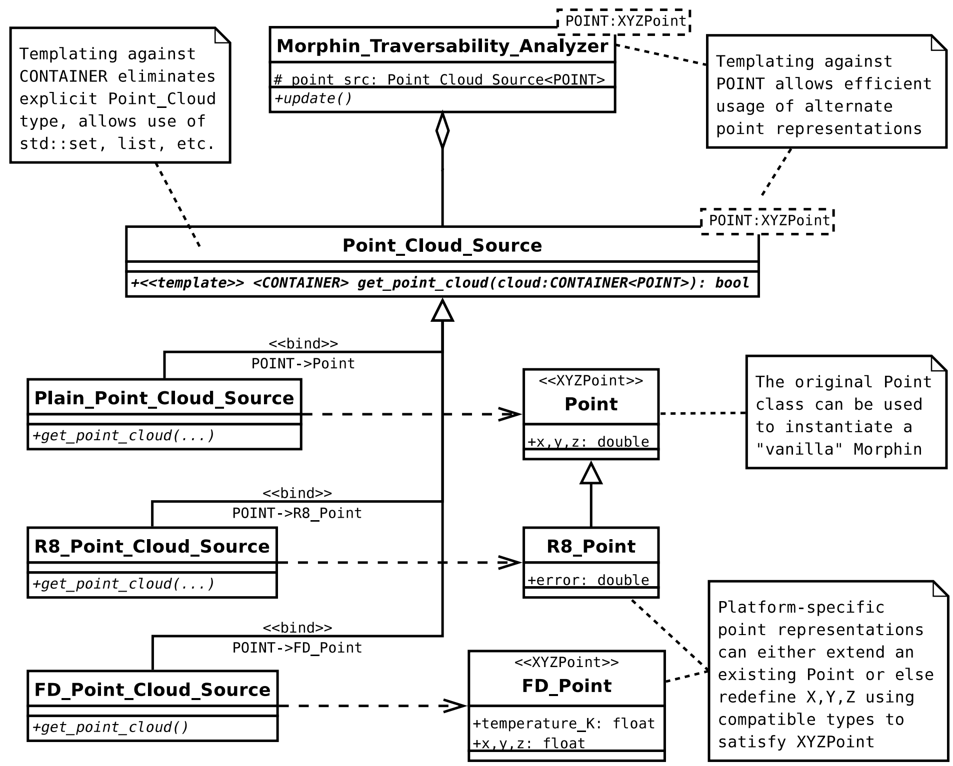


Figure 9.11: Template-based accommodation of alternate point representations for Morphin "point sources"

The alternate design in Figure 9.11 replaces the specific dependency on the `Point` class with a more generic dependency on the `XYZPoint` stereotype[3]. This stereotype, which represents the *primary* data for the Morphin algorithm, can be satisfied either by inheriting from a "basic" point representation, such as with `R8_Point`, or else by direct declaration, such with `FD_Point` in Figure 9.11. The `add_points()` method of the `Morphin` class, along with the `add_point()` method of the `Plane_Fit_Moments` class would be similarly templated, fully decoupling the Morphin algorithm from any specific "point" representation, as shown in Figure 9.12.

This usage of templates is a variation on theme of "mixins" used to introduce supplemental data into the intermediate data types for the OO redesign of the Merge Planner

[3]In template programming, a "stereotype" can be thought of as specifying what "must" be provided in order use the template. In this case, the "XYZPoint" stereotype specifies that the templated "POINT" class can only be satisfied by types that include public, scalar (int, float, double, etc.) members named "x", "y", and "z".

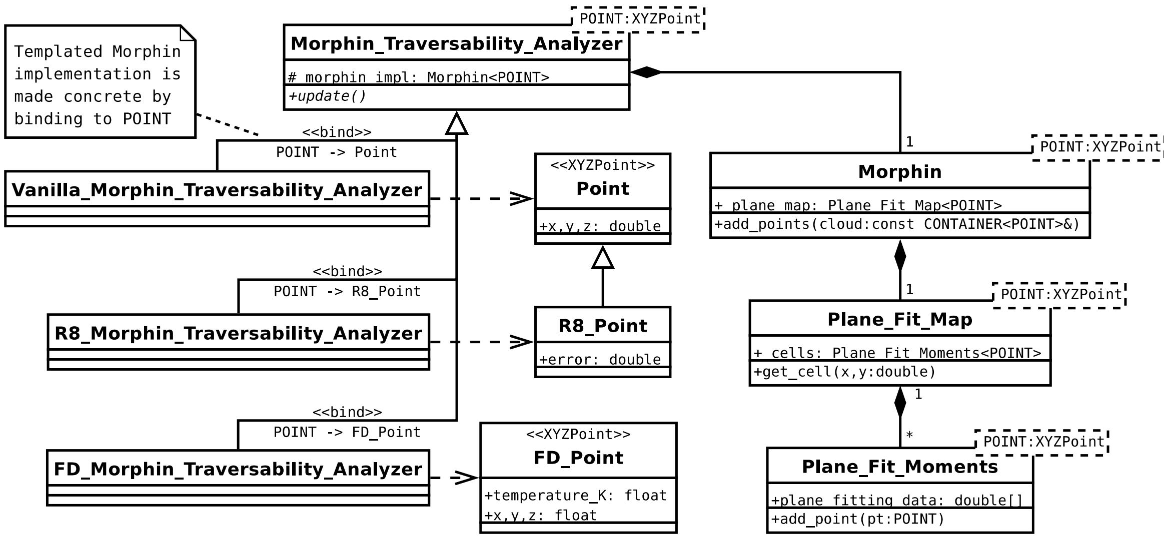


Figure 9.12: Template-based accommodation of alternate point representations for the Morphin Algorithm

in Section 5.4. In fact, Figure 5.14 highlights the usage of type definitions ("typedefs") as pseudo-templates, where the alternative was to template the `OOMergePlanner` class against type extensions for each of three intermediate data types.

In a sense, these variations on the `Point` data types can be thought of as "intermediate" in the broader scope of the `Traversability_Analyzer` role, highlighting the issues of recursion and granularity of design discussed in previous chapters. From this perspective, it makes sense that the OO technique described in Chapter 4 would be directly applicable here, given the existing usage of OO design techniques in CLARAty.

It follows that the proposed AO approach that would be similarly applicable, such as by "slicing" supplemental data into the `Point` class, but doing so would have effects beyond the the Morphin implementation, such as to include effects on the stereo vision or other algorithms that process raw sensor data to yield (augmented) point cloud data. While certainly a valid and interesting application of this methodology, this would move the discussion well beyond scope of the Morphin algorithm, and is thus not pursued further here. Still, this suggests a broader applicability of the *primary* vs. *supplemental* methodology across many levels of the CLARAty hierarchy that could be the basis for interesting future work.

The remainder of this discussion focuses on the `Morphin` class and its subsidiary classes, and the application of the AO and OO techniques proposed in Chapter 4 in order to expose adaptability interfaces and bind supplemental effects to the Morphin algorithm.

Adaptability Interfaces

As highlighted in Chapter 8, relative to introducing novel input data to existing software, the contents of an effective adaptation interface must consider the effects of both existing and possible future supplemental data. In the limit, this requires a degree of prescience that can only be attained through copious amounts of experience and luck in order to determine the "best" such interface. In practice, however, a reasonably good adaptation interface can be derived by combining existing supplemental effects with a few likely candidates for future adaptation, such as the temperature, vegetation, or feature-of-interest examples discussed above.

In this case, "error" information is the only pre-existing supplemental datum, and its effects, which are discussed in more detail in Section 9.3 can be summarized as:

ERR.1 Modify the plane-fitting algorithm by associating a unique weight to each point, computed as $1.0/\mathrm{error}^2$.

ERR.2 Extend the `Plane_Fit_Moments` class to include new members and methods for representing and manipulating error data when aggregated into a plane.

ERR.3 Extend the output interface of the `Plane_Fit_Moments` to expose a new derived result, "quality" for use by downstream processing.

ERR.4 Incorporate the "quality" value into the "goodness" computation. This is ill-specified because the corresponding functionality was not present in the CLARAty implementation of Morphin, presumably excised due to irrelevance.

Taking a more general viewpoint, the first three effects provide reasonable coverage of what might happen to supplemental data as a "point" is incorporated into a "plane". That is, an adaptability interface for the plane-fitting process could cover a broad range of supplemental data by specifying:

PF.V.1 The computation of the weight assigned to a given point shall be extensible to allow application of supplemental data.

PF.V.2 There shall be a means to introduce new member data in the `Plane_Fit_Moments` class to hold additional configuration values and/or for aggregated results of supplemental data.

PF.V.3 The methods that allow plane data to be combined and separated, such as the addition and subtraction operators in Figure 9.7, shall be extensible so as to allow preservation of semantics for aggregated data such as introduced through **PF.V.2**.

PF.V.4 There shall be a means of extending the output interface of the `Plane_Fit_Moments` class to provide a means of propagating novel results, such as based aggregated data mentioned in **PF.V.2**, to downstream consumers.

The fourth effect of the error datum, **ERR.4**, although ill-specified, can be merged with the discussion above to yield a similar adaptability requirements the conversion of plane-fitting data into the actual "goodness" representation.

GC.V.1 The computation of the pitch and roll hazards shall be extensible such that alternate thresholds for maxim allowable angles may be substituted as a function of intermediate results made available through **PF.V.4**.

GC.V.2 The computation of the each of the pitch, roll and traversal hazards shall be extensible such that the overall outputs may be scaled as a function of intermediate results made available through **PF.V.4**.

GC.V.3 The determination of the maximum hazard value shall be extensible to include additional hazards as may be necessary to represent thermal, radiation, or other non-geometric hazards to the platform or mission.

When combined with the plane-fitting adaptability interface prescribed by **PF.V.1** through **PF.V.4** above, these will allow a wide variety of supplemental data to have valid and meaningful effects on the ultimate "goodness" of an individual cell. Following the detailed design techniques presented in Chapter 4, two such adaptability interfaces are discussed below in Figures 9.13 and 9.15.

OO Design

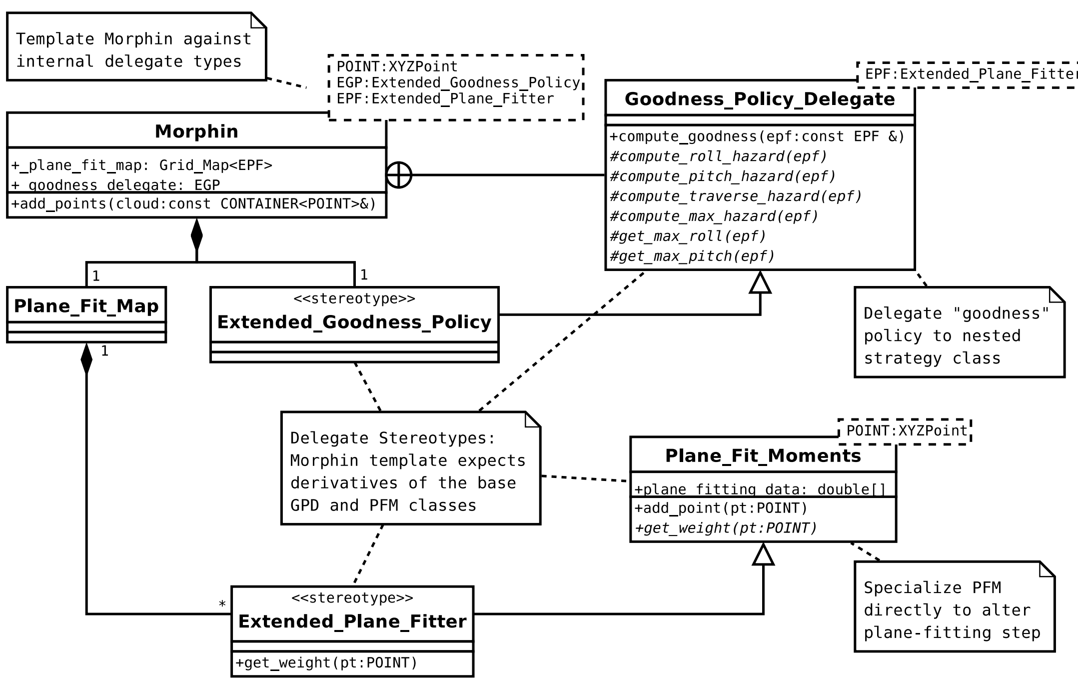


Figure 9.13: Morphin adaptability interface using Object-Oriented delegation.

Rather than generate a separate plane-fitting delegate class, **PF.V.1** through **PF.V.4** are fulfilled by directly specializing the `Plane_Fit_Moments` class. Other than extending this class to include a polymorphic `get_weight()` method, specifically supporting **PF.V.1**, the `PFM` class remains unchanged, as it already does an excellent job of encapsulating the policy for dealing with the *primary* point data.

The rest of the design in Figure 9.13 follows those presented in Chapter 5, specifically by delegating the algorithm for calculating the "goodness" of a `Plane_Fit_Cell` into a nested strategy class, `Goodness_Policy_Delegate`. This class exposes polymorphic methods for the various thresholds and hazard calculations described by **GC.V.1** through **GC.V.3** above. It is also templated against the `Extended_Plane_Fitter` stereotype, discussed below, in order to explicitly accommodate alternate plane-fitting representations

Instead of the previously discussed usage of pseudo templates for the delegate classes, the OO design in Figure 9.13 follows in the footsteps of the "XYZPoint" stereotype from Figure 9.11 and explicitly templates the `Morphin` class against two additional stereotypes:

- `Extended_Plane_Fitter`, which is expected to be inherited from `Plane_Fit_Moments`;
- `Extended_Goodness_Policy`, similarly inherited from `Goodness_Policy_Delegate`.

The corresponding template type-names, `EPF` and `GPD`, are used directly in the declaration of member data in the `Morphin` class, automatically incorporating the extended types into any realization of the Morphin algorithm.

As an example, the original, or "plain" Morphin algorithm can be instantiated as :

```
// A plain Point class that satisfies XYZPoint
class Point { double x,y,z; };

/* Morphin is templated against three types:
 *   - The "point" representation
 *   - The plane-fitting delegate, and
 *   - The "goodness" delegate.
 * The original, or "plain" version may be had as:
 */
typedef Morphin<Point,
                Plane_Fit_Moments<Point>,
                Goodness_Policy_Delegate<Plane_Fit_Moments<Point> >
                > PlainMorphin;
```

Listing 9.1: Instantiation of the "plain" Morphin algorithm according to the design in Figure 9.13

The explicit usage of templates can be somewhat cumbersome, as shown by the nesting of type declarations in Listing 9.1. However, this is not fundamentally different from the previous usage of "typedefs", and the overall adaptability interface is used in much the same way as for the OO designs presented in Chapter 5.

Figure 9.14 shows an example binding of error and temperature supplemental effects, discussed above, to the OO adaptability interface. This example assumes a volcano-exploring robot, perhaps a descendant of the original "Dante"[60] , that is outfitted with an array of sensors that can provide error and temperature information to supplement the Morphin algorithm beyond the *primary* Cartesian point data. This data is encoded in the platform-specific `DantePoint` class, which adds "error" and "temperature" members to the default "XYZPoint" stereotype.

The effects of the "error" data, which are analogous to **ERR.1** through **ERR.4** above, are implemented across two classes, `EPF_Error_Effects` and `EGP_Error_Effects`. The first of these overrides the `get_weight` method of `Plane_Fit_Moments` to implement **ERR.1**, and introduces a `compute_quality()` method to fulfill **ERR.2** and **ERR.3**. This method

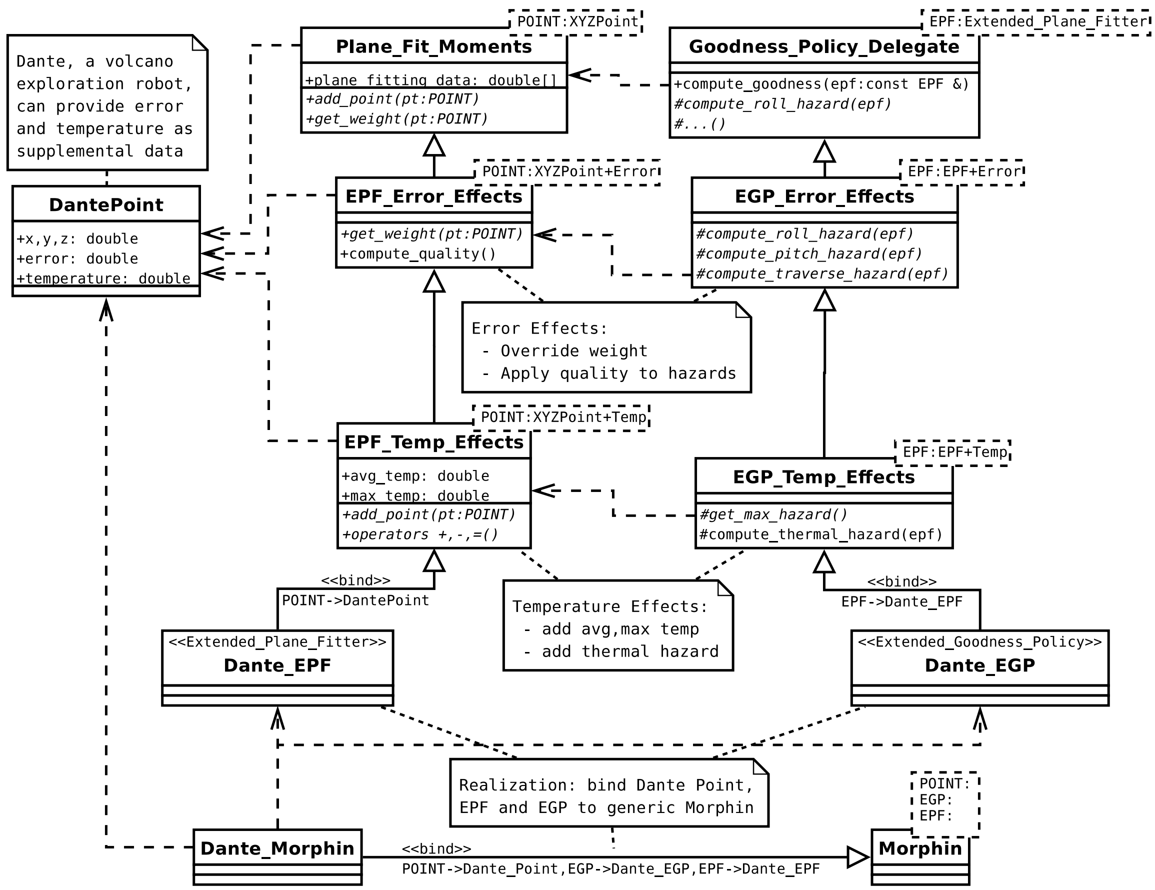


Figure 9.14: Morphin supplemental effects using Object-Oriented delegation.

is then used in `EGP_Error_Effects` to affect the outputs of the roll, pitch, and traverse hazards in fulfillment of **ERR.4**.

There are similar extension classes for the "temperature" data, the first of which, `EPF_Temp_Effects`, augments the `add_point` method of `Plane_Fit_Moments` to include a calculation of average and maximum temperature as points are added to a plane. The second class, `EGP_Temp_Effects`, then uses these values to calculate a thermal hazard, such as to avoid terrain that is "too hot" for the robot to traverse, and augments `get_max_hazard()` to include this in the determination of the "worst" hazard.

Other effects discussed above, such as for vegetation detection or other supplemental data, may be introduced in a similar manner. Although the more explicit use of templates causes this design to appear more convoluted, the overall dependency structure is the same as the OO designs in Chapter 5. That is, the effects of each supplemental datum are isolated in individual classes that specialize base functionality, and these classes are explicitly composed to form the final delegates. In a broad sense, it follows that similar tradeoffs of source code size for improvements in "concern diffusion" and "option value" can be expected, and, accordingly, that this design will enhance the overall adaptability of the Morphin algorithm.

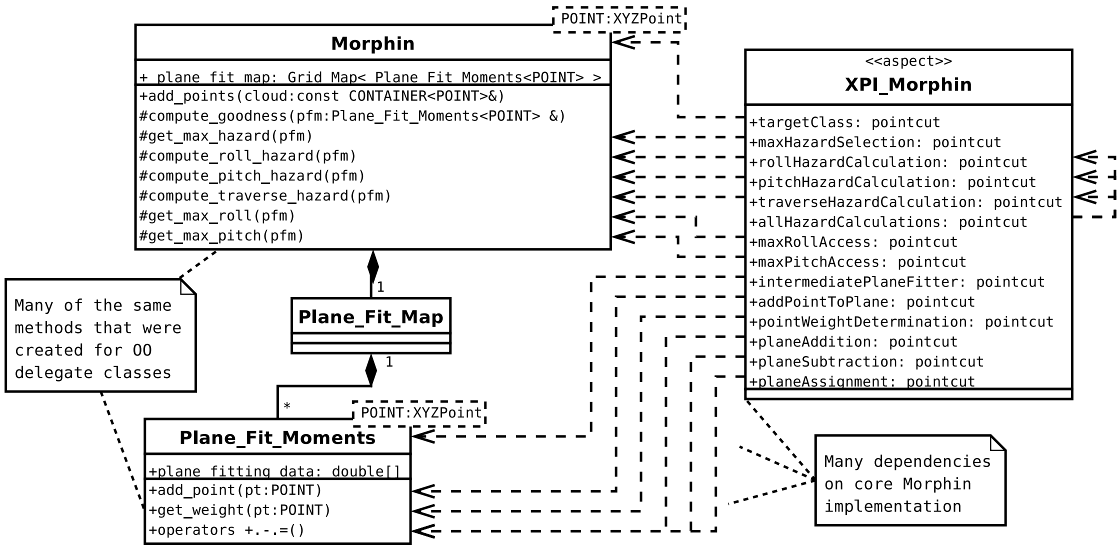


Figure 9.15: Morphin adaptability interface using a Crosscutting Programming Interface (XPI)

AO Design

The analogous AO design, shown in Figure 9.15, leaves the class and template arrangement from Figure 9.10 untouched, and adds an XPI in the same style as the AO designs in Chapter 5. As was the case for those designs, a certain amount of method-level refactoring in the `Morphin` and `Plane_Fit_Moments` classes would be necessary to expose valid join points for AO introduction. These methods are basically the same as those that were delegated to the `Goodness_Policy_Delegate` class in the OO design above, except that they remain within the `Morphin` class, yielding a somewhat more coherent design.

Otherwise, the contents of the XPI are determined in accordance to the variability requirements discussed above, including exposing the `intermediatePlaneFitter` pointcut to allow augmentation of the contents of the `Plane_Fit_Moments` class. Also, as was done for the various relevance tests in the Precedence Estimator in Section 5.3, an aggregated pointcut, `allHazardCalculations`, is declared in order to allow broad effects to be applied more concisely. For example, a policy of increasing all hazard values by some small amount, such as to represent a poor-quality terrain estimate, may be applied to all three hazard calculations simultaneously through this pointcut.

This usage of `allHazardCalculations` is illustrated in Figure 9.16, alongside the other effects of the "error" and "temperature" supplemental data discussed for the OO design above. Herein, the usage of *around()* and *after()* advice, along with the introduction of *slice classes* is nearly identical to the AO designs in Chapter 5. Most importantly, the supplemental effects are more coherently grouped by supplemental datum than in the OO design, accommodating the absence or alteration of a given supplemental datum via the substitution or exclusion of a single aspect.

The tradeoffs for this smaller, more concise representation are the brittle dependencies from `XPI_Morphin` to the underlying `Morphin` class in Figure 9.15, along with the overall

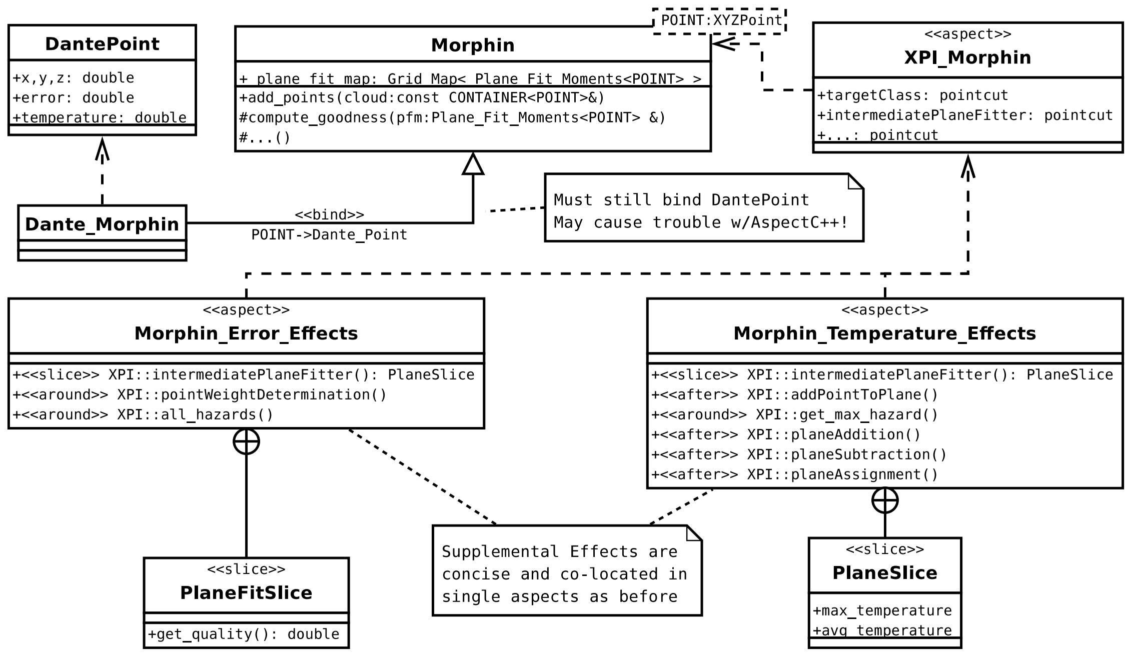


Figure 9.16: Morphin supplemental effects applied using AO introduction.

dependency on prototypical AO technology. The latter issue is hinted at in Figure 9.16, in that the specific usage of templates in CLARAty, such as binding the `DantePoint` representation to the `Morphin` class, may not be supported by the AspectC++ compiler at this time. That is, the AspectC++ compiler is currently unable to weave advice into pure template classes[57], limiting the technical feasibility of the AO design presented above. Still, this compiler remains under active development, and these issues are likely to be resolved in future releases. At that point, a more thorough analysis of the interplay between template-based and aspect-based adaptations would be an interesting path of further investigation.

9.5 Summary

This chapter has presented a design-level case study in applying the *primary* vs. *supplemental* methodology proposed by this thesis to the Morphin terrain analysis algorithm as implemented in the Coupled Layer Architecture for Robot Autonomy. Following the insights gleaned in the more detailed work already done on autonomous driving behaviors, several alternate designs for Morphin have been proposed to enhance the adaptability of the corresponding software artifacts relative to additional, altered, or absent data as outlined in Chapter 2.

These designs begin with a simple refactoring of the Morphin algorithm, independent of more advanced AO or OO design techniques, that increases the adaptability of the existing implementation by realigning its structure to emphasize the path that input data take on the way to the generation of the ultimate "goodness" output. An extraneous intermediate result, the `CSpace_Traversability` representation, was eliminated, along with the corresponding stage in the Morphin processing pipeline, reducing the amount of work that would be necessary to propagate supplemental data in the `Point` representation to where it may have the most effective influence on the "goodness" results.

Following this initial redesign, the AO and OO techniques presented in Chapter 4 were applied, yielding adaptability interfaces and encapsulation of supplemental effects similar to those presented for the autonomous driving software in Chapter 5. These similarities imply tradeoffs of raw code size for reductions in "diffusion" and enhancements to "option value" similar to those presented in Chapters 6 and 7, suggesting similar gains in the ability to accommodate future adaptations as discussed in Chapter 8.

Although these results favor the AO approach over the OO approach, the former may have technical issues, and the latter is highly consistent with existing patterns already deployed in CLARAty. Combined with the fact that earlier results show that both AO and OO approaches will reduce diffusion and add value to existing designs, this leaves the determination of the "best" technique to the judgement of the designer.

Beyond the detailed designs, the most interesting result was the identification of an existing supplemental "error" datum in the original Morphin implementation. The similarity of the effects of "error" data the Morphin algorithm to those identified in autonomous driving software supports the claim that the issues of adaptability outlined in Chapter 2 are common and under-treated phenomena in advanced robotic software. When combined with the straightforward generation of an adaptability interface, and the ease with which other effects, such as for temperature data, might be added through that interface, this complementary case study supports the broad applicability of the methodology proposed by this thesis.

Chapter 10

Summary and Conclusions

This thesis explores the idea that, in order to enhance the *reusability* of advanced robotic software, it is necessary to directly address the *adaptability* of that software to suit the specific needs and capabilities of individual robotic systems. Rather than focusing on increasingly generic or otherwise all-encompassing data representations, as is the norm for the robotics community, this work instead embraces the idea that the sheer variability in robotic systems cannot be captured by a single data representation. The consequences of this, namely that an algorithm must be adapted to some combination of *additional*, *absent*, and *altered* input data as it is "ported" from one platform to the next, are pursued as first-class design concerns, yielding significant insights into the nature of platform-specific effects on otherwise generic algorithms, and how those effects might be treated in a modular manner, as novel contributions to the field.

10.1 *Primary* Contributions

The critical insight offered by this work is that many, if not most, variations in platform-specific capabilities can be framed as subtle enhancements, or *supplements* to the default, or *core* functionality of a given robotic algorithm. Although each such enhancement will be a unique binding of platform-specific capabilities to algorithm-specific context, the overall influence on a given algorithm will be largely restricted to the modification of a few specific parameters, policies, or other *points of variability*.

To enhance reusability across a wide variety of robotic platforms, this thesis proposes a novel distinction between the *primary* data that a robot *must* provide to enable the *core* functionality of an algorithm, and the *supplemental* data that a robot *may* provide to enhance that algorithm in platform- and mission-specific ways. A corresponding methodology is also introduced whereby the *primary* data comprise the traditional input interface for a stable, reusable implementation of the *core* algorithm, which is complemented by a dedicated *adaptation interface* that allows the external application of platform-specific *supplemental effects*.

This methodology is evaluated in a detailed case study on existing software for autonomous driving behaviors, focusing on refactoring three distinct components according to each of two advanced software design techniques that allow the modular treatment of *supplemental effects* as described above. The resulting artifacts were analyzed using two established metrics that capture critical issues in software adaptability: how difficult it is to understand a given piece of software ("Concern Diffusion"), and how well the design of that software accommodates changes by modular substitution, as opposed to invasive modification ("Net Option Value").

The combined results of these analyses show that both traditional Object-Oriented (OO), and more modern Aspect-Oriented (AO), techniques can be used to enhance the adaptability of robotic software to changes in the content or semantics of platform-specific *supplemental data*. Of these, the AO techniques appear to more naturally suit the challenges of separating supplemental effects from a given core algorithm, but the tradeoff for using them is that the technology underlying AO methodology is still largely prototypical, and may introduce an unacceptable level of risk to a project.

In addition to these advanced design techniques, the work in this thesis also uncovered several recurring patterns in the design of *core* robotic algorithms that can make it more difficult to accommodate platform-specific data. These issues, which are explored in greater detail in Appendix A, can be summarized as three simple, if somewhat counter-intuitive, guidelines that can be followed, even in the absence of more advanced AO or OO techniques, to increase the adaptability of an algorithm to platform-specific data:

1. **Propagate input data types as deeply as possible.** If presented with an abstract input data type, such as the `Point` or `MovingObstacle` classes discussed in this thesis, it is better to pass that external representation as deeply as possible into your algorithm than to go to great lengths for the simple sake of "insulation" from that external type. Whether prematurely "breaking out" the external type into its constituent elements, such as for the explicit `add_point(x,y,z)` variations, or else "translating" it into some ephemeral internal representation, these "insulating" barriers create extra work to propagate any *additional* platform-specific data that may present itself in the future. Instead, keep the core algorithm "close" to the original input representation, which will "automatically" propagate that *additional* data to the points where it can have the most meaningful effects.

2. **Account for the possibility of future supplemental data in internal data types.** Where internal representations make more sense, such as for collecting individual candidate obstacles into an aggregated `MergeObstacle`, or adding individual `Points` to a plane-fitting cell, keep in mind that the input obstacles, points, etc., may be augmented with *supplemental* data in the future, and be sure to provide a mechanism for "keeping track" of that data. This can be accomplished either by maintaining references to the original inputs, as done for the foremost and rearmost `MovingObstacle` instances in Section A.4, or else by providing a way to extend the aggregated type to include *additional* data of its own, such as for the average and maximum temperature values discussed in Section 9.4.

3. **Use small, well-named functions for critical data interpretation.** Allow the concepts of "supplemental effects" and "likely points of variability" to guide the method-level decomposition of the core algorithm by pulling platform-specific interpretations out into small, separate functions, instead of leaving them deeply embedded in larger ones. Following #1 above, and the enumeration of common points of variability below, try to frame those small methods as *questions about* a candidate point,

obstacle, etc., instead of simple accessors for some internal (configurable) parameter. For example, small methods such as:

```
double MyAlgo::getMaximumThresholdFor(const MovingObstacle &mo)
```

... will be much easier to identify and understand than two or three lines of logic embedded in a 200-line method, and it will also "automatically" make any *supplemental* moving obstacle data available at that point of reasoning, in case some future *additional* data are relevant to the "maximum threshold" policy. As an added benefit, this will poise the algorithm for more advanced treatment of supplemental effects, such as the OO or AO techniques described in this thesis, should they prove worthwhile at a later time.

Beyond these issues of detailed design and analysis, several candidate augmentations of the urban driving components to include vehicle-to-vehicle communications data were explored in order to test the expressiveness of their adaptation interfaces, gleaning several useful insights into likely points of variability that may be identified in other algorithms. Many of these focus on issues of "confidence" in the *primary* data, including:

1. *Whether* a given candidate point, obstacle, etc., should be included in further computation, such as simply ignoring vehicles for yield calculations if they do not have a "strong lane association" (**PE.S.1**);

2. *How* a given candidate is to be included in further calculation, such as by modulating the degree of "trust" in an obstacle's velocity measurement.

These issues of "confidence" in products of a robot's perception system are clearly relevant to a wide range of robotic algorithms, as, except under the idealism of "perfect perception", all robots are bound by imperfect sensors that only provide a partial view of the surrounding environment. In order to operate effectively on any one robot, many algorithms require a certain "knowledge" of the underlying sensors and modeling techniques that can only be informed by including platform-specific data, such as the RADAR-backed "isMoving" state for candidate moving obstacles, which suggests a broad applicability of the methodology presented in this thesis. This is supported by the complementary case study in Chapter 9, which readily identified similar issues in terrain analysis software for planetary rovers.

In both autonomous driving and planetary exploration, there were also several algorithm-specific points of variability, such as whether a candidate obstacle requires a "courtesy gap" in highway merge calculations, or the various contributors to the "goodness" of a patch of terrain, that will rely more heavily on the judgement and experience of the designer to identify for inclusion in an adaptation interface. Still, these fall under a more general category of answering "How should this candidate {obstacle, point, pixel, etc.} be treated differently?" that are often exposed as configurable gains, thresholds or other parameters in existing systems, which provides an excellent starting point for identifying such esoteric points of variability in other robotic algorithms.

The duality to this is that these categories of "likely" points of variability also inform the distinction between *primary* vs. *supplemental* data, beyond the issues of "intersection vs. union" and "semantic depth" discussed in Chapter 2. That is, if a datum's influence on a given algorithm can be isolated to one or more typical points of variability, including:

1. **Intrinsic "relevance" tests**, such as the use of the "observed-moving" flag to help identify "relevant" traffic at an intersection;

2. **Context-specific parameter substitution**, such as adjusting the degree of "pessimism" to apply to an obstacle's estimated position or speed, substituting an alternate threshold for "observed-moving" vehicles, or determining the weight to assign to an individual point in a "cloud";

3. **Replacement of expensive "default" functionality** with an analogue that is more efficient, but also possibly more brittle, such as bypassing expensive geometric "in-lane" tests with a simple lookup in the "lane associations" supplemental datum.

... then it is *at least possible* to treat that datum as *supplemental*, regardless of whether or not it is "expected" to be available on other robots. That is, considering candidate data in terms of their effects on a given algorithm will provide a lower bound on the set of *supplemental* data, which can be combined with the upper bound on *primary* data provided by the "intersection" of all platform capabilities to yield a comparatively narrow band of candidates that are left to the designer's judgement to classify as to whether they warrant *primary* or *supplemental* treatment.

This thesis informs that judgement through the detailed analysis and discussion of specific examples in existing systems, allowing designers to come to their own conclusions as to whether the future benefits of treating an individual datum as *supplemental* will outweigh the expected costs of doing so.

10.2 *Supplemental* Contributions

Beyond exploring and demonstrating the *primary* contributions of this thesis, the detailed case study of autonomous driving software also generated two *supplemental* contributions to the software engineering community.

Detailed Case Study: Adaptability "In the Small"

Even though small-scale adaptation is a (or even *the*) recurring challenge for software practitioners, the difficulty of gleaning general results through analysis of inherently subjective problems means that issues of adaptability "in the small" are only sporadically covered by the software engineering community. To this fledgling pool, this thesis contributes a detailed case study of such small-scale adaptations in the largely untouched context of advanced robotic software, including a better understanding of the applicability of "concern diffusion" metrics, and the "Net Option Value" model to small-scale issues of software design and adaptability. In particular, the sensitivity analysis of the NOV model in Chapter 7 is more detailed than any similar analysis in identified published works, and it includes novel insights and suggests future paths of research that are being considered for submission to one or more software design venues.

Detailed Case Study: "Weakly Invasive Aspects"

To the Aspect-Oriented software community in particular, this thesis contributes a catalog of aspect-oriented refactorings that will help inform ongoing research into both the general applicability of aspect-oriented techniques and specific efforts to classify aspects according to the nature of their effect on a system. In the latter case, the concept of a "supplemental effect" described in this thesis resonates with issues of "weakly" vs. "strongly" invasive aspects[28]. The categories of supplemental effects described above, and the implied "points of variation" they correspond to, could inform the identification of "weakly invasive" aspects in other problem domains, and is also being considered for submission to an appropriate venue.

10.3 Future Work

Effort Models for Small-Scale Software Modification

As hinted at in the discussion of "supplemental" contributions above, the diffusion and NOV models employed in this thesis only provide a partial picture of software "adaptability", and more research is clearly warranted into the development of "cost" or "effort" models for such small-scale modifications. That is, existing effort models, such as COCOMO[7] are largely based on corporation-level source code and financial data, providing predictive models that convert KSLOC (thousands of source lines of code) into coarse units of effort, such as "person-months". These models explicitly regress past the types of detailed modifications addressed in this thesis, making their applicability to such small-scale efforts unclear. The inherently subjective issues at this level of detail, such as a programmer's familiarity with the algorithm, specific programming style, and overall development and domain experience, pose significant barriers to generalized small-scale effort models that will be difficult to overcome, but would nevertheless be a valuable path of future research.

Connections to Graphical and Model-Based Programming Environments

There is a growing trend toward more graphical, or "model-based" development environments that focus on describing the flow of data between functional modules, as opposed to "traditional" programming at the level of "source-code", as was the case for the components examined in this thesis. While many of the more tedious "source-level" problems of message passing and synchronization disappear at this "higher" level, the problem of having to "break open" existing functional modules in order to incorporate platform-specific data remains. That is, the "policies" that are typically affected by supplemental data can be thought of as "micro-models" for obstacle thresholds, culling policies, etc., that require the same types of surgical modification as for the more "traditional" source-code-based approaches evaluated in this thesis.

Adjunct to the work on autonomous driving components discussed above, initial inroads have been made as to how supplemental effects may be applied to such graphical models, and a simple example, based on the issue of introducing a "has V2V" flag into a Distance Keeping algorithm (similar to **CX.4**) is shown in Figure 10.1:

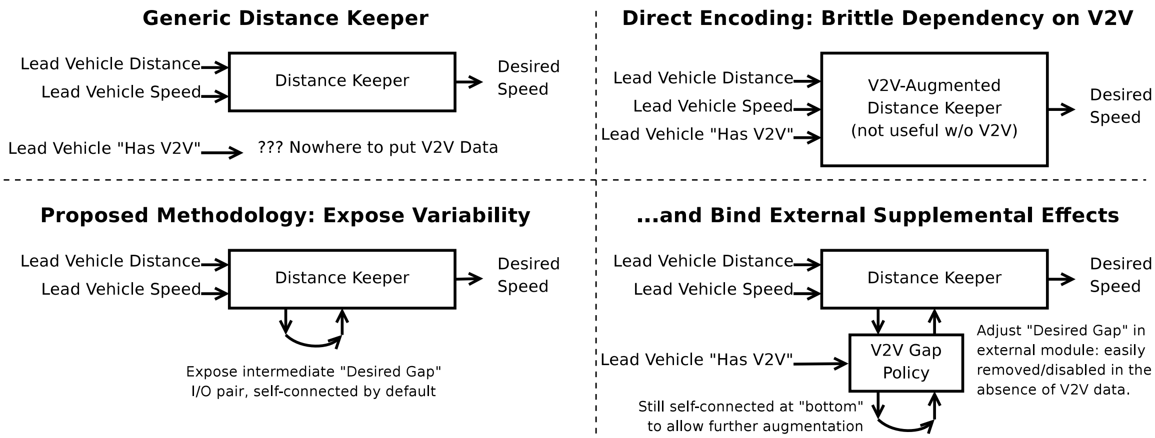


Figure 10.1: Example application of supplemental "V2V" effects using a graphical, model-based development environment, such as Simulink$^{\text{TM}}$.

Here, a simple "Distance Keeper" functional module takes the lead-vehicle distance and speed as inputs and computes a desired speed that will maintain a safe following distance. When presented with a new datum, such as whether the lead vehicle "Has V2V", the Distance Keeper must be "opened up" to include a new "input pin" for the V2V data and to embed the associated effects. The resulting component (Figure 10.1, top-right) is then strongly bound to the V2V input, and must be maintained separately from the more generic version of the Distance Keeper.

To avoid this "opening up" for further *additional* data, the methodology proposed by this thesis could be applied by exposing an otherwise "internal" data connection in the Distance Keeper, representing the "desired gap" for the lead vehicle, so that it may be augmented according to some external policy (Figure 10.1, bottom-left). This would then be used by a dedicated "V2V Effects" module to adjust the desired gap as necessary, leaving the more generic functionality in the core Distance Keeper module untouched (Figure 10.1, bottom-right). In this manner, many such detailed policies can be exposed for augmentation by *supplemental* data, and the guidelines listed above for identifying "likely" points of variation are directly applicable to help determine which "internal" data connections to expose.

It is important to remember, however, that this is only a simplistic example, and the detailed consequences of applying this or other similar techniques to large-scale software models are not immediately clear. As such, application and detailed analysis of these techniques in the context of larger, more complex models, ideally to a set of related models that have been adapted to multiple platforms from a common root, would make an excellent path of future research.

Beyond Autonomous Driving (and Planetary Roving)

Although the results in this thesis support a broad applicability of the proposed *primary* vs. *supplemental* methodology, it is important to continue to identify and catalog supplemental effects in other systems in order to broaden our understanding of typical platform-specific variations of otherwise generic algorithms. This will further assist future designers in identifying when, where, and how to apply the techniques described in this thesis to make it easier for other developers to rapidly adapt advanced robotic algorithms to the detailed capabilities of new robotic systems.

Appendix A

Designing Core Algorithms to Consider Supplemental Effects

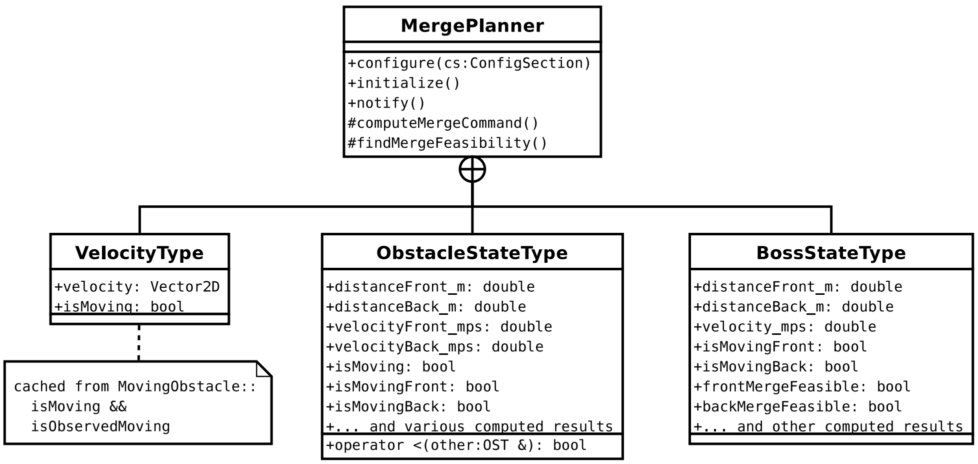
This appendix explores recurring themes in advanced robotic software that, while useful and intuitive from the perspective of "insulating" algorithms from external data representations, can also increase the difficulty of introducing supplemental effects, even given a dedicated "adaptation interface" as proposed by this thesis. These issues were first identified in the usage of intermediate data types for the Merge Planner in Chapter 5, then as a "higher-level" architectural issue regarding supplemental "lead vehicle" data in Chapter 8, and lastly in the CLARAty implementation of Morphin in Chapter 9. In each instance, the need to propagate supplemental data to "downstream" consumers compelled a significant rearrangement of the corresponding *core* algorithms to better support the application of supplemental effects.

The detailed refactoring of the original Merge Planner design, presented below, exposes each of these issues in turn and proposes incremental design changes that preserve functionality and efficiency, while also eliminating key barriers to the incorporation of supplemental data. The insights underlying each incremental change are then distilled into more general guidelines for designing other algorithms to better accommodate supplemental data, which, even in the absence of the more advanced design techniques discussed in this thesis, will enhance the adaptability of those algorithms by making them more understandable and reducing the amount of "extra" work that must be done to propagate platform-specific data to the places where they can have meaningful effects.

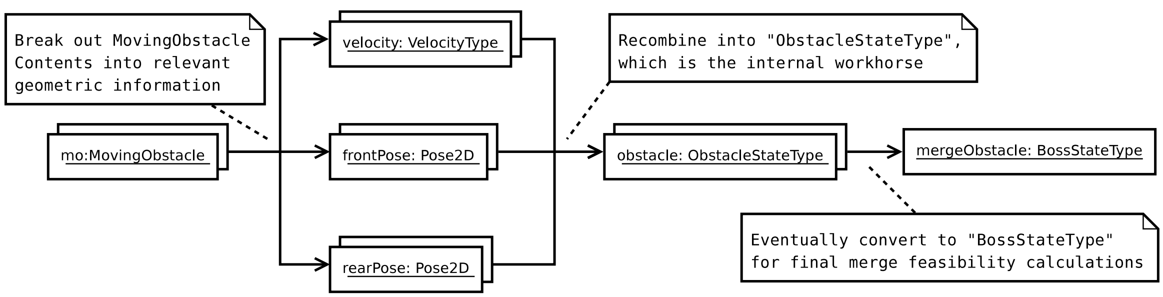
A.1 Original Merge Planner: Review

The original Merge Planner design is repeated in Figure A.1 for reference, highlighting both the declaration and usage of the three intermediate data types as part of the Merge Planner's processing pipeline. This form of "batch" or "staged" processing is common in robotic (and many other types of) software, as it helps break down large calculations into more manageable pieces that are "insulated" from one another by a stable intermediate data representation.

While this pattern enhances understandability and reusability relative to a fixed input specification, the decomposition of an algorithm in this manner also explicitly creates a



(a) Class Collaboration



(b) Data Flow

Figure A.1: Original Merge Planner design, reproduced from Chapter 5 for reference.

sequence of “barriers” to the inclusion of *supplemental data* as described by this thesis. That is, if a novel input datum is presented at the first stage of the processing pipeline, it will often be necessary to augment each intermediate data representation to include that datum, and to extend each stage of processing to include rules for propagating that new datum to each subsequent stage. As discussed in the main body of this thesis, this leads to a great deal of “extra” work to be performed in order to make any given supplemental datum available at the appropriate stage of processing.

Interestingly, this echoes an observation that was made early in the development of advanced robotic architectures[54], which pointed out that the then-modern trend toward layered planning architectures meant either:

1. Sacrificing everything unique and interesting about a platform for the sake of generality, or

2. Enduring a great deal of “extra work” to punch through the layers of abstraction to propagate platform-specific details all the way to the “highest-level” planners.

Their proposed solution, which was counter to the prevailing intuition at the time, was to collapse the layers into as few as possible and focus on specifying detailed “constraints” to an otherwise highly generic “planning engine”. This is highly analogous to, if more narrowly scoped than, the issue of *core* algorithm and *supplemental* effects explored by this thesis, and, at least from a certain perspective, their solution is applicable to this context as well. That is, collapsing long processing pipelines and focusing, somewhat counter-intuitively, on propagating external data types as deeply as possible will reduce the amount of "extra" work necessary to apply supplemental effects to a given core algorithm.

A.2 Insulation from External Data Types

The first stage of the Merge Planners processing pipeline in Figure A.1b represents an initial "breaking-out" of the contents of the `MovingObstacle` representation in to its constituent elements. This reflect the conventional wisdom that it is valuable to insulate an algorithm from abstract external representations by focusing on the most elemental forms of the algorithms input data. In terms of generality, this is indeed a valuable first step, as it makes the data handled by the algorithm more explicit, and provides an easy means of adapting to syntactic changes in external class, method and member names, which often require tedious modifications, but otherwise have no real effect on the actual semantics of the input data[1]. From the perspective of accommodating novel inputs, however, this requires supplemental data to be "broken out" as well, which can lead to a combinatoric explosion of input methods, such as for the variations of `add_point()` in Morphin, or else will require augmentation of intermediate types, as was the case for Merge Planner.

In the case of the Merge Planner, the initial breaking out of the external `MovingObstacle` representation is immediately followed by recomposing those individual elementals, including the intermediate `VelocityType` representation, into the internal `ObstacleStateType` representation, discussed in Section A.4 below. Thereafter, the corresponding instances of `VelocityType` and all other "broken out" data are destroyed, and have no further influence on the Merge Planner's core algorithm.

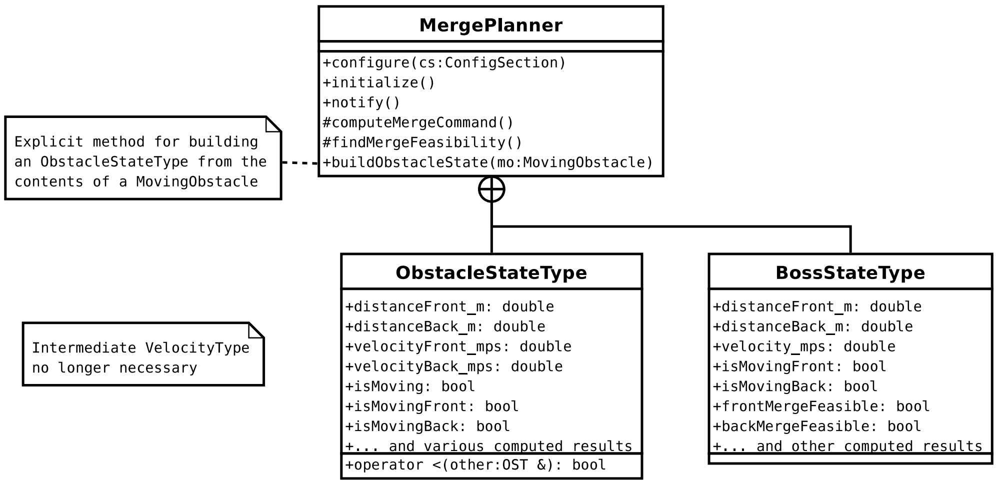
Moreover, there is a one-to-one mapping from `MovingObstacle` instances to the recombined `ObstacleStateType` instances, suggesting that the intermediate `VelocityType` can be eliminated by populating the `ObstacleStateType` directly from the `MovingObstacle` representation. Figure A.2 shows the corresponding removal of the nested `VelocityType` class, the introduction of a dedicated `buildObstacleStateType` method, and the elimination of the "break-out" stage of the original processing pipeline.

A.3 Ephemeral Data Aggregators

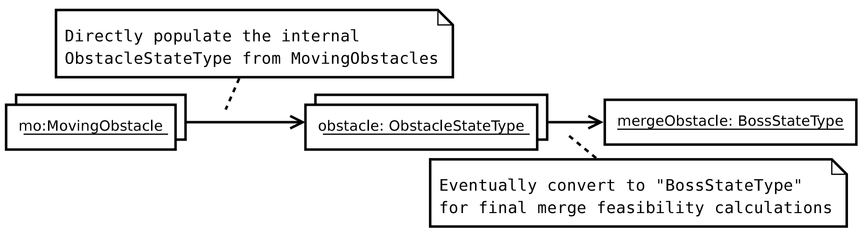
As a complement to the issue of "breaking out" an input representation, there is also a recurring tendency to translate and/or collect intermediate results into temporary data structures that satisfy the requirements of some "downstream" stage of processing[2]. These are immediately destroyed, have no other use in the system, and are barriers to the propagation of supplemental data, suggesting that they too should be considered for elimination.

[1] Consider, for example, the proliferation of geometric utility libraries, containing class representations for Vectors, Points, Polygons, etc, that all have the same underlying meaning, but whose specific representation, such as "pt.x" vs. "pt.x()" vs. "pt[0]", differs. These impose tedious, and usually inefficient, translations from one representation to another just to "glue" two otherwise compatible components together.

[2] Ironically, these are often "broken out" again before being consumed within that next stage.



(a) Class Collaboration



(b) Data Flow

Figure A.2: Eliminating premature "break-out" of the `MovingObstacle` representation.

Not all such translations can be eliminated, such as when converting data representations to make use of proprietary or otherwise "black-box" software components, but in some cases, these conversions are performed for the sake of expediency or efficiency, and can be eliminated by reworking "downstream" stages to make more direct use of "upstream" results. This was the case for the intermediate `CSpace_Traversability` representation, discussed in Chapter 9, which was used to cache several "expensive" plane-fitting calculations for use by the downstream "goodness" calculation. Closer inspection revealed that it would be possible, if somewhat labor-intensive, to bypass this intermediate result and rework the "goodness" calculation to make more direct use of the plane-fitting functionality, caching "expensive" results for itself (or not) as necessary.

In the case of the Merge Planner, the usage of the intermediate `BossStateType` is a more mundane matter of time-constrained development than computational efficiency. That is, the `BossStateType` was an earlier form of the `ObstacleStateType` representation, and there was simply not enough time to properly refactor all of the "older" elements of the Merge Planner to use the "newer" obstacle representation. As with the `CSpace_Traversability` representation, there would be nontrivial effort involved, but such effort would pay dividends in future adaptability by further reducing the number of "barriers" that would have to be perforated by supplemental data. Doing this simplifies the Merge Planner implementation to a single intermediate representation, directly populated by the contents of the `MovingObstacle` representation, as shown in Figure A.3.

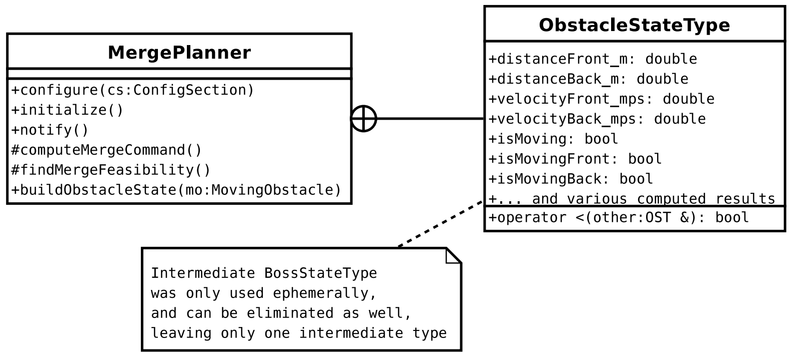


Figure A.3: Eliminating the ephemeral usage of the `BossStateType` representation.

A.4 Augmentation of Internal Data Representations

In some cases, however, there are critical stages of processing in an algorithm, such as the fitting of individual points to a plane in the Morphin algorithm, whose solution intrinsically requires an intermediate data representation. This is the case for the Merge Planner's intermediate `ObstacleStateType`, which includes functionality for sorting obstacles in order along a given lane, combining adjacent obstacles that are "too close" to merge between, and caching derived "merge parameters", all of which are unique and critical to the merge-planning problem.

This class, which might be more aptly-named `MergeObstacle`, is the workhorse of the Merge Planner's core algorithm, and its usage greatly simplifies downstream policies for determining which merge opportunities are "feasible", and to eventually identify the one that is the "best". As with the `Plane_Fit_Moments` class in the Morphin algorithm, there is no way to extract this class from the merge planning algorithm, so it remains as a barrier for propagation of supplemental data to the downstream "merge feasibility" calculations.

This barrier may be overcome by allowing augmentation of this intermediate type, such as following the OO or AO designs presented in Chapter 4, and the results presented in this thesis show that this is an effective approach to take. However, a more straightforward solution would be to simply augment the `MergeObstacle` representation to keep track of its constituent `MovingObstacle` instances.

This was not an option for the the `Plane_Fit_Moments` representation due to possibility of having to keep track of thousands of individual `Point` instances in a system that was expected to be significantly resource-constrained. For the Merge Planner, however, there are no such resource constraints, and there are expected to be relatively few (i.e., less than 10) `MovingObstacle` instances to keep track of, so incorporating them into the `MergeObstacle` representation may be a viable alternative.

Upon closer inspection, especially of the semantics and usage of the `isMovingFront` and `isMovingBack` data, reveals that it would only be necessary to keep track of the foremost and rearmost obstacles in this manner. Moreover, the policy for keeping track of them could be embedded in the core algorithm, eliminating the need to augment the "adjacent obstacle combination" policy as before, and thus "automatically" propagating any supplemental data in the `MovingObstacle` representation to the downstream "merge feasibility" policies.

Doing so further simplifies the Merge Planner's algorithm to exclude the explicit caching of the intermediate "isMoving" states, replacing them with more useful references to the foremost (`frontMO`) and rearmost (`rearMO`) `MovingObstacle` instances that contribute to a given `MergeObstacle`[3], as shown in Figure A.4.

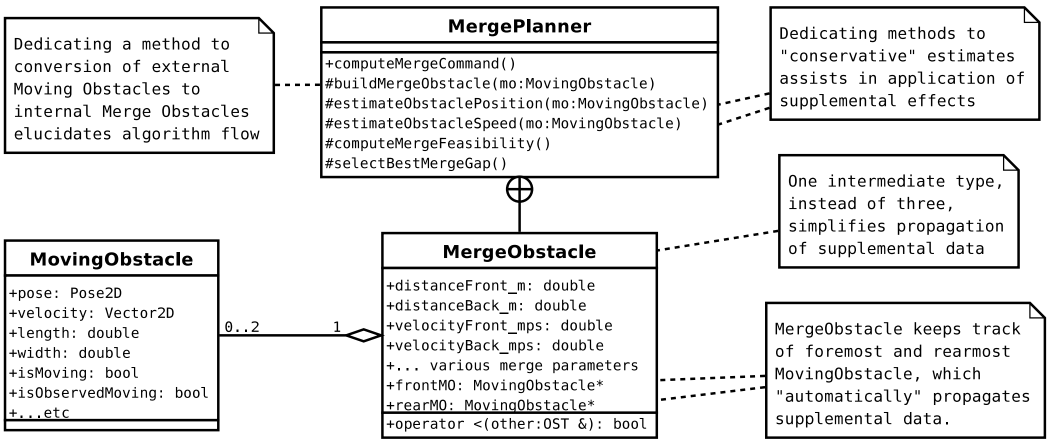


Figure A.4: Renaming `ObstacleStateType` to `MergeObstacle` and incorporating constituent `MovingObstacle` instances to "automatically" propagate supplemental data therein.

The critical benefit of this final simplification is that all supplemental data in the `MovingObstacle` representation are "automatically" available to downstream policies through the `frontMO` and `rearMO` members of `MergeObstacle`. Thus, *all* of the additional effort put into introducing, deriving, and propagating the intermediate "isMoving" states, or any other such supplemental data, is no longer necessary.

A.5 Summary

By avoiding unnecessary "insulative" translations, and focusing instead on propagating external data representations as deeply into an algorithm as possible, the lessons embedded in the above discussion can be applied to many other advanced robotic algorithms to make them more readily adaptable to platform-specific data. Even if the more advanced techniques described in this thesis are not warranted, or possible, during the initial development of an algorithm, following a few simple guidelines can reduce the amount of "extra" work necessary to incorporate novel input data into a that algorithm:

1. Avoid unnecessary and/or premature "breaking out" of external data representations into their constituent elements: if those external representations are later augmented to include *additional* data, then a separate channel for that data must be created in order to propagate its contents to downstream consumers.

[3]These would initially be set by `computeMergeObstacle` to point to the same `MovingObstacle` instance, but would eventually come to reference different MovingObstacle instances through the `combineAdjacentObstacles` method.

2. Avoid the use of "ephemeral" data types for the sake of reusing "older" or "legacy" code as-is: updating that older code[4] to make more direct use of newer or otherwise upstream data types will pay dividends in future adaptability.

3. In general, avoid long processing pipelines: the closer the output results of an algorithm are to its input data representation, the less work will have to be done to augment intermediate stages to propagate future *supplemental* data. There is clearly a tradeoff to be made here, between the benefits of pipelined processing and the difficulty of adapting individual stages to novel data, which is at least partially informed by this thesis and would certainly be a valuable topic of future research.

4. Where intermediate data representations make sense or cannot otherwise be avoided, keep a list (if resources limitations permit) of the input data that contribute to a given instance of that representation: this will "automatically" propagate any future supplements to that input representation to the next stage of processing.

5. Where resources do not permit such a list, carefully document the fact that that intermediate representation may have to be extended, along with the critical transformations thereof that would have to be augmented to include policies for future *supplemental* data: these will be valuable guides for future development, and may be used to seed more advanced adaptation interfaces at a later date.

[4]This is obviously not possible for "black box" legacy components, or for other imported components that cannot (for technical and/or political reasons) be modified to better accommodate a particular platform. This highlights one of the critical issues with "black box" software reuse: that it is often desirable to tune a reused software component in ways that are not accessible through the otherwise opaque interface. In such cases, the mechanisms for translating to and from the "legacy" interface should be thoroughly encapsulated and the limitations thereof should be clearly documented so that downstream developers (and upstream managers!) can easily identify the constraints imposed by the legacy components.

Appendix B

Aspect-Oriented Programming

A canonical example from the Aspect-Oriented Programming (AOP) literature[33] is the implementation of thread safety in an otherwise straightforward class. As a concrete example, consider a simple class for adding two numbers, such as shown in Listing B.1:

```
class Adder {
 public:

  // mutator for the x_ member
  void setX(int x) {
    x_ = x;
  }

  // mutator for the y_ member
  void setY(int y) {
    y_ = y;
  }

  // accessor for the sum of x_ and y_
  int getSum() {
    return x_ + y_;
  }

 private:
  int x_,y_;
};
```

Listing B.1: Target class for AspectC++ example

This class is concise, easy to understand, and completely focused on the core "concerns" of storing two integers and presenting their sum. Additional functionality is required, however, to guarantee safe access to these members from multiple threads. In the language of software requirements, this thread-safety "concern" may be specified as:

> Every method that interacts with the internal state of a given class shall lock a class-specific mutex at the beginning and release it at the end.

The implementation of the thread-safety concern using classical techniques requires a mutex object (of type `MutexType` in these examples) to be added to the `Adder` class, and two identical lines of code be added to each method in Listing B.1. This *scatters* the implementation of thread safety through each method and *tangles* the core functionality of each method with the thread-safety concern as highlighted by comments in Listing B.2:

```
class Adder {
 public:

  // mutator for the x_ member
  void setX(int x) {
    myMutex_.lock();      // thread safety concern
    x_ = x;
    myMutex_.release();   // thread safety concern
  }

  // mutator for the y_ member
  void setY(int y) {
    myMutex_.lock();      // thread safety concern
    y_ = y;
    myMutex_.release();   // thread safety concern
  }

  // accessor for the sum of x_ and y_
  int getSum() {
    myMutex_.lock();      // thread safety concern
    int ret =  x_ + y_;
    myMutex_.release();   // thread safety concern
    return ret;
  }

 private:
  MutexType myMutex_;     // thread safety concern
  int x_,y_;
};
```

Listing B.2: Target class for AspectC++ example, with direct introduction of mutual exclusion

The AO community refers to thread-safety and similar concerns as "crosscutting concerns", as their implementation "cuts across" the dominant source structure of class and method declarations. This makes the `Adder` class more difficult to understand by "polluting" the core functionality with the thread-safety concern, and conversely makes the thread-safety concern harder to maintain, as it is spread across the entire `Adder` class definition.

In contrast, AOP allows the use of *point-cuts* that may be specified using wild-card characters, *advice* directives and the *join-point* model[1] to encapsulate such "crosscutting" concerns separately from the core functionality. That is, the AO implementation of the thread-safety concern leaves the `Adder` class definition in Listing B.1 untouched and implements the thread-safety concern in a separate module, called an *aspect*, as shown in Listing B.3.

```
aspect MutualExclusion {
 public:
  // "slice" the mutex into the target class
  advice "Adder" : slice class {
    MutexType myMutex_;
  };

  // describe the join points
  pointcut accessors() =
    execution("% Adder::get(...)");

  pointcut mutators() =
    execution("% Adder::set(...)");

  // do this around each join-point
  advice accessors() || mutators() : around() {
    // lock the mutex for the class
    tjp->that()->myMutex_.lock();
    // execute the original code
    tjp->proceed();
    // release the mutex for the class
    tjp->that()->myMutex_.release();
  }
};
```

Listing B.3: Implementation of mutual exclusion using AO techniques

Using the appropriate *aspect weaver*,[2], the advice in the `MutualExclusion` aspect is then triggered at all matching *join points* in the underlying system, which in this case are all methods that match "% Adder::get%(...)" or "% Adder::get%(...)". These declarations use the wild-card character "%" and the ellipsis "..." argument to concisely describe broad groups of methods in the `Adder` class. Encoding thread safety as an aspect promotes the resultant system's:

- Reusability: Thread safety may be added to or removed from this or any other class without interfering with the existing functionality;

[1] The join-point model is a collection of information about the join-point that is programmatically accessible. For AspectC++, every advice directive implicitly contains a variable "tjp" that provides access to this information.

[2] In this case, the syntax is that of AspectC++, available at http://www.aspectc.org

- Modifiability: The specific implementation of thread safety (i.e. `MutexType`) may be altered in exactly one module and re-woven as necessary instead of having to change it in every method of the class;

- Understandability: The actual methods of the `Adder` class are *oblivious* [18] to the presence of a thread safety concern and thus remain focused solely on their specific functionalities. Conversely, the `MutualExclusion` aspect is *oblivious* to the actual contents of `Adder` and only deals with the issue of thread safety.

These are all highly desirable features of a “good” modular decomposition, making AO techniques highly attractive for encapsulating these and similar “crosscutting concerns”.

Appendix C

Concern Listing

C.1 Traffic Estimator

Concern	Description
C	Core algorithm: estimate lead vehicle distance and speed
C.0	Infrastructure, configuration and initialization
C.1	Determine forward path
C.2	Identify vehicles along forward path
C.3	Estimate bumper-bumper distance
C.4	Estimate in-lane speed
C.5	Estimate closest road blockage
V	Exposing and binding algorithmic variability
V.0	Configuration and initialization
V.1	Obstacle “in lane” test
V.2	Obstacle speed estimation
E	Supplemental effects
E.1	Aggregated effects: `MovingObstacle::isMoving`
E.1.2	Require `isMoving` for nonzero speed (**TE.S.1**)
E.2	Aggregated effects: `MovingObstacle::isObservedMoving`
E.2.2	Require `isObservedMoving` for nonzero speed (**TE.S.2**)
E.3	Aggregated effects: `MovingObstacle::isPredicted`
E.3.2	Require `!isPredicted` for negative/oncoming speed (**TE.S.3**)
E.4	Aggregated effects: `MovingObstacle::laneAssociations`
E.4.2	Require `(laneAssociations.size() == 1)` for negative speed (**TE.S.4**)
E.4.1	Replace geometric in-lane test with `laneAssociations` (**TE.S.5**)

Table C.1: Traffic Estimator: Concern Listing for Diffusion Metrics

C.2 Precedence Estimator

Concern	Description
C	Core algorithm: estimate precedence and clearance at intersections
C.0	Infrastructure, configuration and initialization
C.1	Compute intersection occupancy zones
C.2	Test obstacles for occupancy in these zones
C.3	Use order of first occupancy to determine precedence ordering
C.4	Use intersection occupancy to determine intersection clearance
C.5	Calculate yield windows
C.6	Maintain override timeouts for precedence, clearance and gridlock
C.7	Determine boss occupancy of zones
V	Exposing and binding algorithmic variability
V.0	Configuration and initialization
V.1	Critical stages in obstacle set update procedure
V.2	Intersection quiescence test
V.3	Intersection obstacle relevance test
V.4	Exit waypoint obstacle relevance test
V.5	Yield lane obstacle relevance test
E	Supplemental effects
E.0	MaxIgnorableSpeed Requirement (**PE.C.1**)
E.1	Aggregated effects: `MovingObstacle::isObservedMoving`
E.1.1	Require `isObservedMoving` for yield relevance (**PE.S.3**)
E.1.2	Obstacles in intersection with `isObservedMoving` resets intersection override timeout (**PE.S.4**)
E.1.3	Require `isObservedMoving` for all relevance after intersection override timeout has expired (**PE.S.2**)
E.2	Aggregated effects: `MovingObstacle::laneAssociations`
E.2.1	Require non-empty `laneAssociations` for all relevance (**PE.S.1**)

Table C.2: Precedence Estimator: Concern Listing for Diffusion Metrics

C.3 Merge Planner

Concern	Description
C	Core algorithm: identify, synchronize with and merge into the optimum slot
C.0	Infrastructure, configuration and initialization
C.1	Identify obstacles relevant to the current merge scenario
C.2	Translate relevant obstacles into internal data types
C.3	Combine adjacent/overlapping obstacles to account for perception artifacts
C.4	Compute necessary spacing and time-to-merge for each remaining obstacle
C.5	Select "best" feasible merge slot based on time-to-merge and distance to goal
C.6	Synchronize with and merge into "best" merge slot
C.7	Maintain a time-delayed, persistent list of obstacles to account for transient perception errors
V	Exposing and binding algorithmic variability
V.0	Configuration and initialization
V.1	Intermediate obstacle types: extensibility
V.2	Intermediate obstacle types: translation
V.2.1	`MovingObstacle` → `VelocityType`
V.2.2	`VelocityType` → `ObstacleStateType`
V.2.3	`ObstacleStateType` → `BossStateType`
V.3	Adjacent obstacle combination (as `ObstacleStateType`)
V.4	Obstacle speed estimation (as `VelocityType`)
V.5	Obstacle forward gap requirement (as `ObstacleStateType`)
V.6	Obstacle oncoming state determination (as `BossStateType`)
V.7	Obstacle culling distance determination
V.8	Obstacle travel lane determination
E	Supplemental effects
E.0	Intermediate "isMoving" states
E.0.0	Introduction and initialization (**MP.D.1**,**MP.D.2**,**MP.D.3**)
E.0.1	Propagation rules through various types (**MP.D.4**,**MP.D.7**,**MP.D.9**)
E.0.2	Combination rule when adjacent obstacles are condensed (**MP.D.8**)
E.0.3	Require `BossStateType::isMovingFront` for "oncoming" traffic (**MP.D.12**)
E.0.4	Require `ObstacleStateType::isMoving` for courtesy gap (**MP.D.11**)
E.0.5	Require `VelocityType::isMoving` for nonzero obstacle speed (**MP.D.10**)
E.1	Aggregated effects: `MovingObstacle::isMoving`
E.1.1	Require `isMoving` for `VelocityType::isMoving` (**MP.D.5**)
E.2	Aggregated effects: `MovingObstacle::isObservedMoving`
E.1.1	Require `isObservedMoving` for `VelocityType::isMoving` (**MP.D.6**)
E.2.2	Require vehicles with `(isObservedMoving == false)` to be closer (configurable) to the host before consideration in merge calculations (**MP.S.2**)
E.3	Aggregated effects: `MovingObstacle::laneAssociations`
E.3.1	Use `laneAssociations` to determine obstacle travel lane (**MP.S.1**)

Table C.3: Merge Planner: Concern Listing for Diffusion Metrics

Appendix D

Design Structure Matrices

	DE Traffic Estimator		1	2	3	4	5	6
Core Alg.	Estimate Lane Speed	1						
	Select Closest Bumper	2	X					
	Identify Vehicles On Forward Path	3		X				
	Determine Forward Path	4			X			
	Configuration and Initialization	5						
	Class Definition	6	X	X	X	X	X	
Env. Params	Pose	7	X	X	X		X	X
	Velocity	8	X					
	Size	9		X	X			
	Is Moving	10	X					
	Is Observed Moving	11	X					
	Is Predicted	12	X					
	Lane Associations	13	X		X		X	X

Figure D.1: DSM for Traffic Estimator, Direct Encoding Implementation

	OO Traffic Estimator		1	2	3	4	5	6	7	8	9	10	11	12	13	14
SE's	Speed_MovingEffect	1						X								
	Speed_ObservedMovingEffect	2						X								
	Speed_IsPredictedEffect	3						X								
	Speed_LaneAssociationEffect	4						X								
	InLane_LaneAssociationEffect	5							X							
DI's	SpeedCalculationDelegate	6	X	X	X	X					X					
	InLaneTestDelegate	7					X						X			
	GenericDelegate	8	X	X	X	X	X	X	X		X		X			
Core Alg.	Estimate Lane Speed	9														
	Select Closest Bumper	10									X					
	Identify Vehicles On Forward Path	11										X				
	Determine Forward Path	12											X			
	Configuration and Initialization	13														
	Class Definition	14									X	X	X	X	X	
Env. Params	Pose	15									X	X	X		X	X
	Velocity	16									X					
	Size	17										X	X			
	Is Moving	18	X													
	Is Observed Moving	19		X												
	Is Predicted	20			X											
	Lane Associations	21				X	X									

Figure D.2: DSM for Traffic Estimator, Object-Oriented Implementation

	AO Traffic Estimator		1	2	3	4	5	6	7	8	9	10	11	12	13	14
SE's	AOTE_MovingEffects	1														
	AOTE_ObservedMovingEffects	2														
	AOTE_IsPredictedEffects	3														
	AOTE_LaneAssociationEffects	4														
XPI	Lane Speed Calculation	5	X	X	X	X										
	In Lane Test	6				X										
	Target Configuration	7				X										
	Target Slice Class	8				X										
Core Alg.	Estimate Lane Speed	9					X									
	Select Closest Bumper	10									X					
	Identify Vehicles On Forward Path	11						X				X				
	Determine Forward Path	12											X			
	Configuration and Initialization	13							X							
	Class Definition	14								X	X	X	X	X	X	
Env. Params	Pose	15									X	X	X		X	X
	Velocity	16									X					
	Size	17										X	X			
	Is Moving	18	X													
	Is Observed Moving	19		X												
	Is Predicted	20			X											
	Lane Associations	21				X										

Figure D.3: DSM for Traffic Estimator, Aspect-Oriented Implementation

	DE Precedence Estimator		1	2	3	4	5	6	7	8	9
Core Alg.	Maintain Override Timeouts	1		X				X			
	Calculate Yield Window	2	X								
	Determine Intersection Clearance	3	X								
	Determine Precedence Ordering	4	X								
	Boss Occupancy	5	X		X						
	Moving Obstacle Occupancy	6		X	X						
	Build Occupancy Zones	7		X			X	X			
	Configuration and Initialization	8									
	Class Definition	9	X	X	X	X	X	X	X	X	
Env. Params	Pose	10	X	X				X		X	X
	Velocity	11	X	X							
	Size	12		X				X			
	Is Moving	13									
	Is Observed Moving	14	X	X				X		X	X
	Is Predicted	15									
	Lane Associations	16		X				X		X	X

Figure D.4: DSM for Precedence Estimator, Direct Encoding Implementation

	OO Precedence Estimator		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
SE's	Classification_LaneAssociationEffect	1						X												
	Classification_ObservedMovingEffect	2						X												
	UpdateProcess_ObservedMovingEffect	3							X											
	Quiescence_ObservedMovingEffect	4								X										
	ObservedMoving_SharedState	5			X	X														
DI's	ObstacleClassificationDelegate	6	X	X									X				X			
	ObstacleUpdateProcessDelegate	7			X												X			
	IntersectionQuiescenceTestDelegate	8				X						X		X						
	GenericDelegate	9	X	X	X	X	X	X	X	X										
Core Alg.	Maintain Override Timeouts	10											X				X			
	Calculate Yield Window	11										X								
	Determine Intersection Clearnance	12										X								
	Determine Precedence Ordering	13										X								
	Boss Occupancy	14										X		X						
	Moving Obstacle Occupancy	15											X	X						
	Build Occupancy Zones	16											X			X	X			
	Configuration and Initialization	17																		
	Class Definition	18										X	X	X	X	X	X	X	X	
Env. Params	Pose	19										X	X				X		X	X
	Velocity	20										X	X							
	Size	21											X				X			
	Is Moving	22																		
	Is Observed Moving	23		X	X	X	X													
	Is Predicted	24																		
	Lane Associations	25	X																	

Figure D.5: DSM for Precedence Estimator, Object-Oriented Implementation

	AO Precedence Estimator		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
SE's	AOPE_MaxIgnorableSpeedEffect	1																					
	AOPE_LaneAssociationEffects	2	X																				
	AOPE_ObserveMovingEffects	3	X																				
XPI	Obstacle Update Notification	4			X																		
	Obstacle Occupies Intersection	5			X																		
	Intersection Quiescence Test	6			X																		
	Combined Relevance Tests	7	X	X	X																		
	Intersection Obstacle Relevance Test	8							X														
	Stopline Obstacle Relevance Test	9							X														
	Yield Lane Obstacle Relevance Test	10			X				X														
	Target Configuration	11	X	X	X																		
	Target Slice Class	12	X	X	X																		
Core Algorithm	Maintain Override Timeouts	13						X								X				X			
	Calculate Yield Window	14													X								
	Determine Intersection Clearance	15			X										X								
	Determine Precedence Ordering	16													X								
	Boss Occupancy	17													X		X						
	Moving Obstacle Occupancy	18				X	X			X	X	X				X	X						
	Build Occupancy Zones	19														X			X	X			
	Configuration and Initialization	20											X										
	Class Definition	21												X	X	X	X	X	X	X	X	X	
Env. Params	Pose	22													X	X				X		X	X
	Velocity	23													X	X							
	Size	24														X				X			
	Is Moving	25																					
	Is Observed Moving	26			X																		
	Is Predicted	27																					
	Lane Associations	28		X																			

Figure D.6: DSM for Precedence Estimator, Aspect-Oriented Implementation

	DE Merge Planner		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Core Algorithm	Synchronize and Merge	1															
	Identify Optimal Merge Gap	2	X														
	Compute Feasible Merge Gaps	3		X													
	Combine Adjacent Obstacles	4			X												
	Translate To Boss State	5															
	Translate to Obstacle State	6															
	Translate to Velocity Type	7															
	Persistent List of Relevant Obstacles	8															
	Identify Relevant Obstacles	9								X							
	Determine Current Merge Scenario	10									X						
	Intermediate Boss State Type	11			X	X	X										
	Intermediate Obstacle State Type	12					X	X									
	Intermediate Obstacle Velocity Type	13						X	X								
	Configuration and Initialization	14															
	Class Definition	15	X	X	X					X	X	X				X	
Env. Params	Pose	16						X	X		X					X	X
	Velocity	17							X								X
	Size	18						X			X						X
	Is Moving	19							X				X	X	X		
	Is Observed Moving	20							X		X		X	X	X	X	X
	Is Predicted	21															
	Lane Associations	22									X						

Figure D.7: DSM for Merge Planner, Direct Encoding Implementation

	OO Merge Planner		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
Supplemental Effects	OTL_LaneAssociationEffects	1											X																							
	OCR_ObservedMovingEffects	2												X																						
	IMD_ObservedMovingEffects	3													X																					
	IMD_MovingEffects	4													X																					
	IntermediateMovingDelegate.h	5													X																					
	NVT_IntermediateMovingEffects	6														X																				
	VGRT_IntermediateMovingEffects	7															X																			
	VelocityTypeExtension	8			X	X	X											X																		
	ObstacleStateExtension	9					X	X											X																	
	BossStateExtension	10					X		X											X																
Delegate Interfaces	ObstacleTravelLaneDelegate	11	X																																	
	ObstacleCullingRangeDelegate	12		X																																
	IntermediateObstacleTypeDelegate	13			X	X	X																		X	X	X	X								
	NegativeVelocityTestDelegate	14						X																X												
	VehicleGapRequirementDelegate	15							X															X												
	VelocityTypeExtension	16																																		X
	ObstacleStateExtension	17																																		X
	BossStateExtension	18																																		X
	GenericDelegate	19	X	X	X	X	X	X	X																										X	
Core Algorithm	Synchronize and Merge	20																																		
	Identify Optimal Merge Gap	21																				X														
	Compute Feasible Merge Gaps	22																					X													
	Combine Adjacent Obstacles	23																						X												
	Translate To Boss State	24																																		
	Translate to Obstacle State	25																																		
	Translate to Velocity Type	26																																		
	Persistent List of Relevant Obstacles	27																																		
	Identify Relevant Vehicles	28																											X							
	Determine Current Merge Scenario	29																												X						
	Intermediate Boss State Type	30																						X	X	X										
	Intermediate Obstacle State Type	31																								X	X									
	Intermediate Obstacle Velocity Type	32																									X	X								
	Configuration and Initialization	33																																		
	Class Definition	34																				X	X	X					X	X	X				X	
Env. Params	Pose	35																									X	X		X					X	X
	Velocity	36																										X								X
	Size	37																									X			X						X
	Is Moving	38				X																														
	Is Observed Moving	39		X	X																															
	Is Predicted	40																																		
	Lane Associations	41	X																																	

Figure D.8: DSM for Merge Planner, Object-Oriented Implementation

Env. Params							Core Algorithm															Crosscutting Interface (XPI)														SE's				
Lane Associations	Is Predicted	Is Observed Moving	Is Moving	Size	Velocity	Pose	Class Definition	Configuration and Initialization	Intermediate Obstacle Velocity Type	Intermediate Obstacle State Type	Intermediate Boss State Type	Determine Current Merge Scenario	Identify Relevant Vehicles	Persistent List of Relevant Obstacles	Translate to Velocity Type	Translate to Obstacle State	Translate To Boss State	Combine Adjacent Obstacles	Compute Feasible Merge Gaps	Identify Optimal Merge Gap	Synchronize and Merge	Target Class Slice	Configuration and Initialization	Intermediate Velocity Slice	Intermediate Obstacle Slice	Intermediate Boss State Slice	Vehicle Gap Requirement test	Negative Velocity Test	Adjacent Obstacle Combination	Translation: Road Block to "Velocity"	Translation: Mvng Obst to "Velocity"	Translation: "Velocity" to "Obstacle"	Translation: "Obstacle" to "Boss"	Culling Range Determination	Obstacle Lane Determination	AOMP_IntermediateMovingStates	AOMP_ObservedMovingEffects	AOMP_MovingEffects	AOMP_LaneAssociationEffects	AO Merge Planner
40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
X																																			X					1
			X																												X					X				2
		X																				X	X									X			X		X				3
																								X	X	X	X	X	X	X	X	X	X				X	X	X	4
													X																										5
													X																										6
																	X																						7
																X																							8
															X																								9
															X																								10
																		X																					11
																			X																				12
																			X																				13
											X																												14
										X																													15
									X																														16
								X																															17
							X																																18
							X													X																				19
							X												X																				20
							X				X						X																						21
											X																												22
										X	X																													23
				X		X			X	X																														24
					X	X			X																															25
							X						X																										26
				X		X	X				X																													27
							X																																28
																																								29
																																								30
																																								31
						X	X																																	32
				X	X	X																																	33

Figure D.9: DSM for Merge Planner, Aspect-Oriented Implementation

Appendix E

The Unified Modeling Language (UML)

This appendix provides reference diagrams that clarify the usage of UML[48] in this thesis, which is based in large part on the first chapter of [44]. In particular, some diagrams make use of obscure UML notations, such as the "nested" class relationship in Figure E.5, or otherwise deviate from standard UML, such as to represent aspects[56]. In general, the diagrams take dependency relationships much more precisely than normal, especially when describing method-level dependencies that are necessary for fine-grained analysis.

Example figures are populated with several notes to elucidate the meaning of various symbols, often negating the need for accompanying explanatory prose beyond the caption. Example C++ or AspectC++ syntax is provided in accompanying listings to further elucidate the meaning of the diagrams, and each figure-listing pair is placed on a separate page, beginning on page 188, to ensure maximum clarity.

E.1 UML Basics

Figure E.1: Example UML diagram, introducing the "note" or "comment" box ("I am a comment") and the most basic representation of classes (named boxes with multiple compartments) and "objects" which are instances of a given class.

```
// I am a comment about MyClass
class MyClass {};   // declare a class
MyClass myInstance; // declare an instance of MyClass
MyClass myArray[16]; // devlare an array of MyClass-es
```

Listing E.1: C++ syntax for Figure E.1

E.2 Class Contents and Inheritance

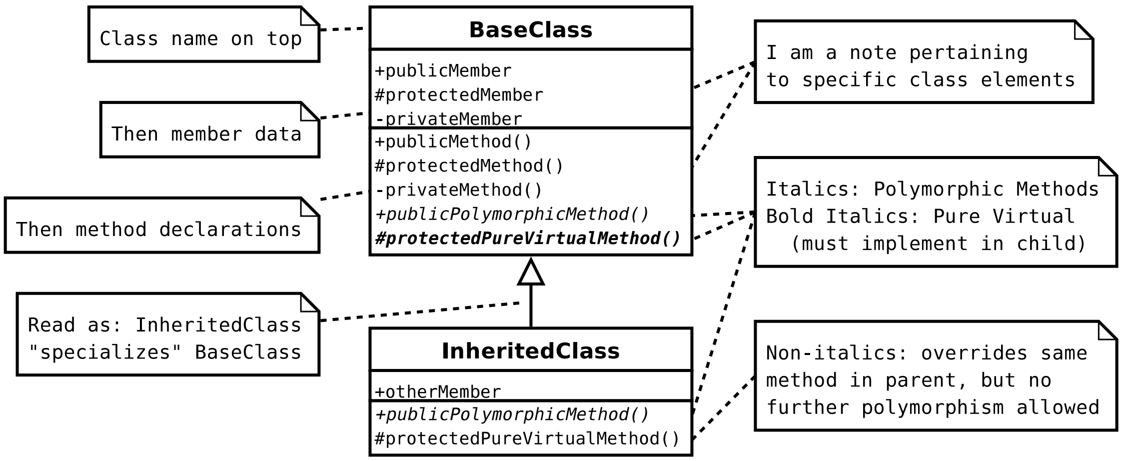


Figure E.2: Example UML diagram, introducing the standard arrangement of class names, members and methods, and the notation for inheritance, or *specialization*, which is the open-ended arrow from `InheritedClass` to `BaseClass`.

```
class BaseClass {
  public:
    bool publicMember;
    void publicMethod();
    virtual void publicPolymorphicMethod();
  protected:
    bool protectedMember;
    void protectedMethod();
    virtual void protectedPureVirtualMethod() = 0;
  private:
    bool privateMember;
    void privateMethod();
};

class InheritedClass : public BaseClass {
  public:
    bool otherMember;
    // override parent functionality: children can also override
    virtual void publicPolymorphicMethod();
  protected:
    // override parent functionality: no further polymorphism
    void protectedPureVirtualMethod();
};
```

Listing E.2: C++ syntax for Figure E.2

E.3 Inter-class Dependencies

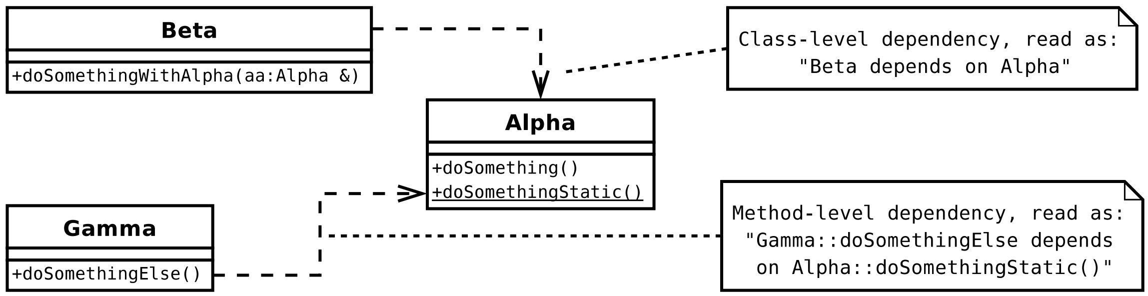


Figure E.3: Example UML diagram, demonstrating both class-level and element-level dependencies. Note that this is not a common usage of UML, but is necessary to demonstrate the calling patterns amongst interrelated methods.

```
class Alpha {
 public:
  void doSomething();
  static void doSomethingStatic();
};

class Beta {
  void doSomethingWithAlpha(Alpha &aa) {
    // poke Alpha
    aa.doSomething();
    // and do other stuff
  }
};

class Gamma {
  void doSomethingElse() {
    // do Gamma stuff
    // ... which includes a static side effect in Alpha
    Alpha::doSomethingStatic();
  }
};
```

Listing E.3: C++ syntax for Figure E.3

E.4 Aggregation

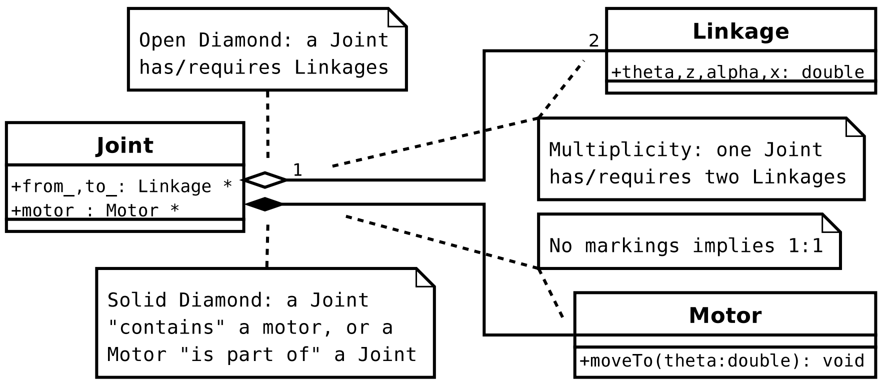


Figure E.4: Example UML diagram, demonstrating different degrees of aggregation between classes: has (undecorated) vs. requires/owns (open diamond) vs. contains/composes (solid diamond). Note that these are highly interrelated, with semantic overlap between.

```
class Linkage {
  double theta, z, alpha, a;
};

class Motor {
  void moveTo(double theta);
};

class Joint {
  // aggregated classes are typically created elsewhere,
  // and passed to the container at construction
  Joint(Linkage *from, Linkage *to):
   from_ (from), to_ (to) {
     motor_ = new Motor();  // composed instances are created...
  }

  ~Joint() {
    delete motor_; // ... and destroyed with their containers
  }

  Linkage *from_, *to_;
  Motor *motor_;
};
```

Listing E.4: C++ syntax for Figure E.4

E.5 Nested Typing

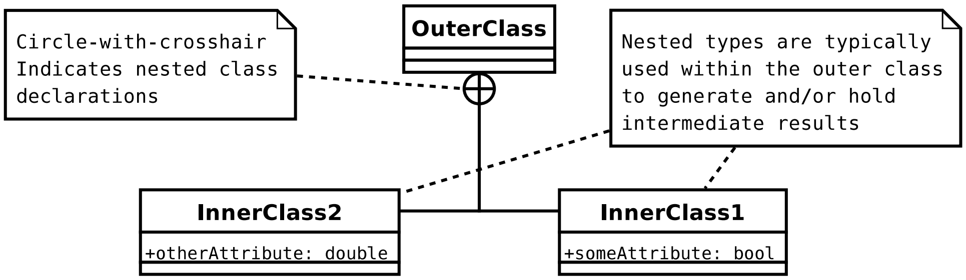


Figure E.5: Example UML diagram, demonstrating nested type declarations.

```
class OuterClass
{
  public:
  class InnerClass1
  {
    bool someAttribute;
  };

  class InnerClass2
  {
    double otherAttribute;
  };
};
```

Listing E.5: C++ syntax for Figure E.5

E.6 Templates

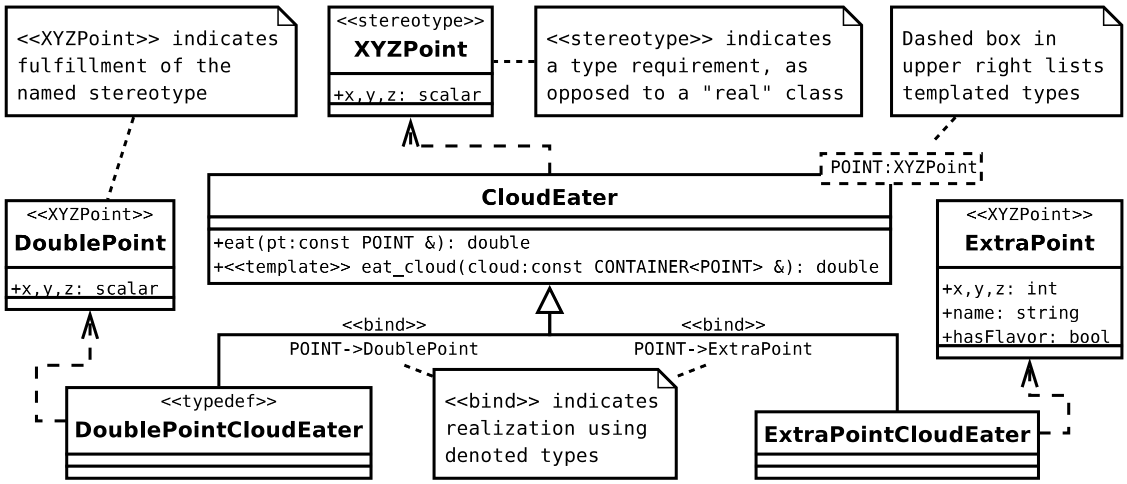


Figure E.6: Example UML diagram, demonstrating template class notation.

```cpp
// POINT must satisfy the XYZPoint stereotype by having
// public scalar member data named "x", "y", and "z"
template <class POINT> class CloudEater {
  public:
    // referencing "x", "y" and "z" creates the XYZPoint stereotype
    double eat(const POINT &pt) { return (pt.x + pt.y) * pt.z; }

    // anonymous template eater for a generic CONTAINER of points
    // CONTAINER must support forward iteration, such as std::list
    template <class CONTAINER>
    double eat_cloud(const CONTAINER<POINT> &cloud) {
      double sum = 0.0;
      for(CONTAINER<POINT>::const_iterator ii = cloud.begin();
          ii != cloud.end(); ++ii) { sum += eat(*ii); }
      return sum;
    }
};
// declare a point-class of doubles
class DoublePoint { public: double x,y,z; }
// bind via typedef
typedef CloudEater<DoublePoint> DoublePointCloudEater;

// structs, other scalar types for x,y,z, and arbitrary extras ok
struct ExtraPoint  { int x,y,z; string name; bool hasFlavor; };
// bind via inheritance
class ExtraPointCloudEater : public CloudEater<ExtraPoint> {};
```

Listing E.6: C++ syntax for Figure E.6

E.7 Aspect-Oriented Notation

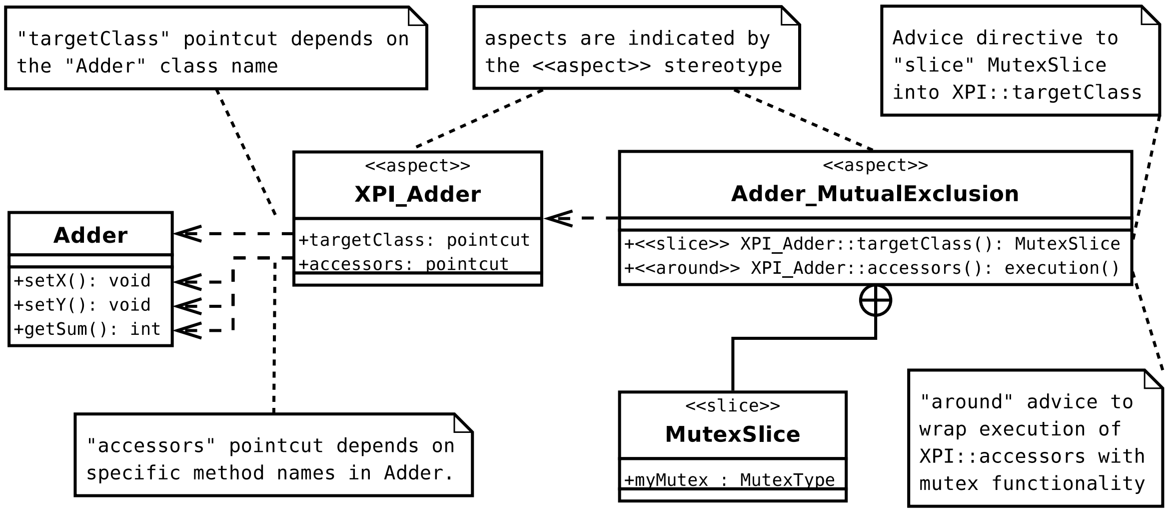


Figure E.7: Example UML diagram, showing aspect-oriented notation from [56], which are consistent with the representation of fine-grained dependencies shown in Figure E.3

```
class Adder {}; // declaration as in Listing A.1

aspect XPI_Adder {
  pointcut targetClass() = "Adder";
  pointcut accessors() = "% Adder::set%(...)" ||
                         "% Adder::get%(...)";
};

aspect Adder_MutualExclusion {
  slice class MutexSlice {
    MutexType myMutex_;
  };

  // slice in the mutex
  advice XPI_Adder::targetClass() : slice MutexSlice;

  advice execution(XPI_Adder::accessors()) :
   around() {
    // do mutual exclusion as in Listing A.3
   }
};
```

Listing E.7: AspectC++ syntax for Figure E.7, see Appendix B for functionality

Bibliography

[1] Jonathan Aldrich. *Open Modules: Modular Reasoning about Advice*, chapter 7, pages 144–168. Springer Berlin/Heidelberg, 2005.

[2] Ronald C. Arkin. *Behavior-Based Robotics (Intelligent Robotics and Autonomous Agents)*. MIT Press, 1998.

[3] Christopher R. Baker and John M. Dolan. Traffic Interaction in the Urban Challenge: Putting Boss on its Best Behavior. In *IEEE/RSJ International Conference on Intelligent RObots and Systems*, pages 1752–1758, 2008.

[4] Christopher R. Baker and John M. Dolan. Street smarts for boss: Behavioral subsystem engineering for the urban challenge. *IEEE/RAS Robotics and Automation Magazine Special Issue on Software Engineering in Robotics*, 16(1):78–87, 2009.

[5] Carliss Y. Baldwin and Kim B. Clark. *Design Rules Vol. I, The Power of Modularity*. MIT Press, Cambridge, MA, 2000.

[6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[7] Barry Boehm, A. Winsor Brown, Ray Madachy, and Ye Yang. A software product line life cycle cost estimation model. *Empirical Software Engineering, International Symposium on*, 0:156–164, 2004.

[8] David Bradley, Ranjith Unnikrishnan, and J. Andrew (Drew) Bagnell. Vegetation detection for driving in complex environments. In *IEEE International Conference on Robotics and Automation*, April 2007.

[9] Rodney A Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[10] Siobhan Clarke and Robert J. Walker. Separating crosscutting concerns across the lifecycle: From composition patterns to aspectj and hyper/j. Technical report, UCD, 2001.

[11] Toby H.J. Collet, Bruce A. MacDonald, and Brian P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proceedings of the Australasian Conference on Robotics and Automation (ACRA), Sydney, Australia*, 2005.

[12] Carle Cote, Yannick Brosseau, Dominic Letourneau, Clement Raievsky, and Francois Michaud. Robotic software integration using MARIE. *International Journal of Advanced Robotic Systems*, 3:55–60, 2006.

[13] Anthony Cowley, Luiz Chaimowicz, and Camillo J. Taylor. Design minimalism in robotics programming. *International Journal of Advanced Robotic Systems*, 3(1):31–36, November 2008.

[14] Kris De Volder, Maurice Glandrup, Siobhán Clarke, and Robert Filman, editors. *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, October 2001.

[15] Dedicated Short Range Communications Technical Committee. Dedicated short range communications (dsrc) message set dictionary. Standards Document J2735, Society of Automotive Engineers, SAE World Headquarters, 400 Commonwealth Drive, Warrendale, PA 15096-0001, USA, 2009.

[16] Defense Advanced Research Projects Agency (DARPA). Urban challenge website, July 2007. http://www.darpa.mil/grandchallenge.

[17] Tara Estlin, Richard Volpe, Issa A.D. Nesnas, Darren Mutz, Forest Fisher, Barbara Engelhardt, and Steve Chien. Decision-making in a robotic architecture for autonomy. In *6th Internation Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS), Montreal Canada*, June 2001.

[18] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000.

[19] Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. AspectC++: Language proposal and prototype implementation. In De Volder et al. [14].

[20] Andreas Gal, Wolfgang Schroeder-Preikschat, and Olaf Spinczyk. Aspectc++: Language proposal and prototype implementation. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[22] Alessandro F. Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos José Pereira de Lucena, and Arndt von Staa. Modularizing design patterns with aspects: A quantitative study. In *Transactions on Aspect-Oriented Software Development I* [47], pages 36–74.

[23] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[24] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60, 2006.

[25] Karl Iagnemma and Martin Buehler, editors. *Special Issue on the DARPA Grand Challenge, Part 1*, volume 23. Wiley, 2006.

[26] Karl Iagnemma and Martin Buehler, editors. *Special Issue on the DARPA Grand Challenge, Part 2*, volume 23. Wiley, 2006.

[27] JAUS. Joint architecture for unmanned systems reference architecture, version 3.2. Technical report, JAUS Working Group, August 13, 2004.

[28] Shmuel Katz. Aspect categories and classes of temporal properties. In *Transactions on Aspect-Oriented Software Development I* [47], pages 106–134.

[29] Stephen Kell. A survey of practical software adaptation techniques. *Journal of Universal Computer Science*, 14(13):2110–2157, 2008.

[30] Alonzo I. Kelly. *RANGER - An Intelligent Predictive Controller for Unmanned Ground Vehicles*. PhD thesis, The Robotics Institute, Carnegie Mellon University, 1994.

[31] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, January 1996.

[32] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[33] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[34] Cristina Videira Lopes and Sushil Bajracharya. Assessing aspect modularizations using design structure matrix and net option value. In *Transactions on Aspect-Oriented Software Development I* [47], pages 1–35.

[35] Cristina Videira Lopes and Sushil Krishna Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2005. ACM. General Chair-Mezini, Mira and Program Chair-Tarr, Peri.

[36] Jun Miura, Motokuni Ito, and Yoshiaki Shira. A three-level control architecture for autonomous vehicle driving in a dynamic and uncertain traffic environment. In *ITS*, pages 706–711, Boston, MA, 1997.

[37] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) Toolkit. In *Proceedings of the International Conference on Intelligent Robots and Systems*, 2003.

[38] Issa Nesnas, Anne Wright, Max Bajracharya, Reid Simmons, Tara Estlin, and Won Soo Kim. CLARAty: An architecture for reusable robotic software. In *Proceedings of SPIE*, 2003.

[39] Issa A. Nesnas, Reid Simmons, Daniel Gaines, Clayton Kunz, Antonio Diaz-Calderon, Tara Estlin, Richard Madison, John Guineau, Michael McHenry, I hsiang Shu, and David Apfelbaum. CLARAty: Challenges and steps toward reusable robotic software. *International Journal of Advanced Robotic Systems*, 3(1):023–030, 2006.

[40] Issa A. D. Nesnas. The claraty project: Coping with hardware and software heterogeneity. *Software Engineering for Experimental Robotics*, 30:31–70, April 2007.

[41] Martin E. Nordberg III. Aspect-oriented dependency inversion. In De Volder et al. [14].

[42] Harold Ossher and Peri Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proc. 23rd Int'l Conf. on Software Engineering*, pages 729–730. IEEE Computer Society, 2001.

[43] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[44] Dan Pilone and Neil Pitman. *UML 2.0 In a Nutshell: A Desktop Quick Reference*. O'Reilly Media, 2005.

[45] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes In C*. Cambridge University Press, second edition, 1999.

[46] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source robot operating system. In *Proceedings of the Open-Source Software workshop at the International Conference on Robotics and Automation (ICRA)*, 2009.

[47] Awais Rashid and Mehmet Aksit, editors. *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*. Springer, 2006.

[48] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[49] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[50] Reid Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings Conference on Intelligent Robotics and Systems*, October 1998.

[51] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[52] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of the 2006 OOPSLA Conference*, New York, NY, USA, 2006. ACM.

[53] Anthony Stentz. Optimal and efficient path planning for part-known environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, 1994.

[54] Anthony (Tony) Stentz and Chuck Thorpe. Against complex architectures. In *Proceedings of the 6th International Symposium on Unmanned Untethered Submersible Technology*, pages 308 – 311, June 1989.

[55] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA, 2001. ACM.

[56] Junichi Suzuki and Yoshikazu Yamamoto. Extending uml with aspects: Aspect support in the design phase. In *3rd Aspect Oriented Workshop at ECOOP*, 1999.

[57] Matthias Urban and Olaf Spinczyk. Aspectc++ compiler manual. Online, August 2010.

[58] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M. N. Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William "Red" Whittaker, Ziv Wolkowicki, and Jason Ziglar. Autonomous Driving in Urban Environments: Boss and the DARPA Urban Challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[59] Christopher Urmson, Reid Simmons, and Issa Nesnas. A generic framework for robotic navigation. In *IEEE Aerospace Conference 2003*, March 2003.

[60] David Wettergreen, Chuck Thorpe, and William (Red) L. Whittaker. Exploring mount erebus by walking robot. *Robotics and Autonomous Systems*, 1993.

[61] David A. Wheeler. sloccount: a tool for counting lines of code in multiple languages, July 2009. http://www.dwheeler.com/sloccount.