Toward Adaptation and Reuse of Advanced Robotic Software

Christopher R. Baker, John M. Dolan, Shige Wang, and Bakhtiar B. Litkouhi

***Abstract*— As robotic software systems become larger and more complex, it is increasingly important to reuse existing software components to control development costs. For relatively simple components, such as perception algorithms and actuation interfaces, this is a reasonably straightforward process, and many excellent frameworks have been developed in recent years that support reuse of such components in novel systems. For more advanced software components, such as for modeling and interacting with the robot's environment, reuse is, however, a more challenging problem. In particular, these advanced algorithms tend to be highly sensitive to the perception and actuation capabilities of the robots they are deployed on, and they often require nontrivial modification to accommodate the specific capabilities of any one robotic system. This work examines the nature of this sensitivity and proposes a novel design methodology for isolating a stable, reusable "core" algorithm from any platform-specific enhancements, or "supplemental" effects. Modern software engineering techniques are used to encapsulate these supplemental effects separately from the core algorithm, allowing platform-specific details to be accommodated by modular substitution instead of direct modification of the "core" component. This methodology is experimentally evaluated on existing software for autonomous driving behaviors, yielding useful insights into the creation of highly adaptable robotic software components.**

I. 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[1], Player/Stage[2], CLARAty[3] and the emerging ROS[4]. These generally focus on reuse by carefully specifying components' expected input and output data such that, as long as all of its specified inputs are available, a given 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 are recognized as undesirable side-effects that are "to be avoided when possible", but there is little direct discussion of what to do when such modifications cannot be avoided. This leaves developers with very little guidance as how to 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].

Still, discussions of the "lessons learned", or "difficulties encountered", in the development and application of component-based architectures can provide insight into some of the specific challenges of adapting robotic algorithms to other systems. For instance, the MARIE[5] framework successfully demonstrated the integration of components from several of the aforementioned toolkits using the Adapter and Mediator patterns[6], but their approach relies on the assumption that the data provided by producing components is semantically identical to the data expected by consuming components. One particular "difficulty encountered" noted by the authors of MARIE is that components often contain "hidden assumptions" about the robots they run on, requiring significant understanding of those components' inner workings, along with detailed knowledge of the original and target platforms, to rectify.

This difficulty is consistent with discussion surrounding CLARAty[7], 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 such components, for which "neither the union of all possible capabilities, nor their intersection"[7] were satisfactory.

Where these and other previous efforts focus on isolating this 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 relevant data that a robot could provide to the underlying algorithm. The proposed al-

This work was supported by the General Motors Autonomous Driving Collaborative Research Lab at Carnegie Mellon University

C. Baker and J. Dolan are with the Robotics Institute at Carnegie Mellon University, Pittsburgh, Pennsylvania, USA {cbaker,jmd}@ri.cmu.edu

B. Litkouhi and S. Wang are with General Motors Global Research and Development, Warren, Michigan, USA {bakhtiar.litkouhi,shige.wang}@gm.com

[1]Large blocks of commented-out or conditionally-compiled source code, such as via `#ifdef MY_ROBOT ... #endif`, are a common symptom of ad-hoc adaptation which can degrade a component's understandability and consequent reusability.

ternative is to classify candidate input data as either *primary* data that enable the *core* algorithm of a given component, or else *supplemental* data that only enhance the *core* algorithm to exploit platform-specific capabilities. The *primary* data are then used to specify a traditional input interface, allowing the *core* algorithm, implemented free of platform-specific enhancements, to be reused on a wide variety of robotic systems. The platform-specific *supplemental effects* are then bound through a dedicated *adaptation interface* that, as opposed to attempting to enumerate all possible *supplemental* data, instead enumerates the ways that the *core* algorithm may be meaningfully adapted to exploit the unique strengths, or compensate for the specific limitations, of any one particular robot.

While this may seem straightforward, the division between *primary* and *supplemental* data, along with the exact composition of the adaptation interface, will be highly algorithm-specific, relying on the judgement and foresight of individual designers to identify "good" versions of each. Moreover, it is not immediately apparent that advanced robotic algorithms can be easily separated into "core" and "supplemental" elements, nor is it clear that the proposed derivation and usage of a complementary "adaptation interface" is a technically feasible solution.

The goals of this work are to inform the judgement of *primary* vs. *supplemental* data through a detailed analysis of supplemental effects in existing software components, and to present and compare two technical approaches to deriving and using adaptation interfaces that can accommodate a wide assortment of platform-specific variations. Together, these will guide the design and implementation of advanced robotic algorithms such that they are both reusable across a wider array of robotic platforms, and also more easily adaptable to the specific details of each system.

This begins in Section II with detailed examples of platform-specific enhancements to otherwise generic algorithms, followed by a more thorough exploration of the proposed proposed *primary* vs. *supplemental* methodology by identifying supplemental effects in existing autonomous driving software in Section III. Section IV follows with a presentation of two technical approaches to modularizing these supplemental effects, and the resulting guidelines for enhancing the adaptability of advanced robotic software are then presented in Section V.

II. On the Nature of Adaptation

Consider the common example of a "point cloud", as used by a wide variety of mapping, localization, terrain analysis, and other advanced robotic algorithms, which may be derived using one or more of LADAR, stereo vision, or other perception techniques. Classical approaches focus on reusability by specifying highly generic input representations, as typified by the Cartesian coordinates in the `Point` class in Figure 1. Advanced algorithms are then implemented within coherent modules, represented by the `TerrainAnalysis` class, which uses the contents of the `Point` representation to generate an obstacle map.

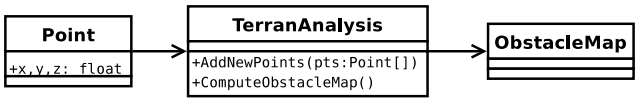

Fig. 1. Simplified data-flow model common to most component-based architectures. Notation: UML[8]

As long as the `Point` class is representative of the information that a given robot can provide, the `TerrainAnalysis` class can be reused without modification. The trouble with robots, as alluded to above, 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 datum would require introduction of an "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 "error" information, but may instead provide temperature readings, such as from stereoscopic thermal imagers, or an indication of the presence of vegetation through more sophisticated analysis techniques. These data could also be relevant to the `TerrainAnalysis` algorithm, but are not compatible with the existing "error" data, yielding the *semantic mismatch* shown in Figure 2.

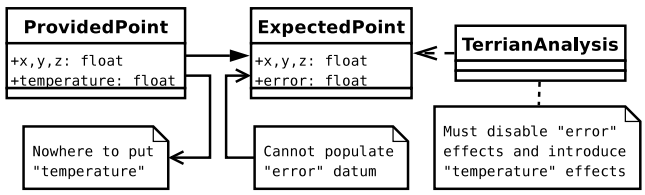

Fig. 2. Semantic mismatch between an input representation that expects points to be annotated with error information, and a platform that provides temperature readings instead.

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 or disabled depending on the detailed capabilities of each specific robot. In the general case, 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"[9] 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. This can cause any combination of:

1) **Additional** data to be incorporated into algorithms to enhance performance, safety, etc.;
2) **Absent** data that can no longer be derived by the system, requiring excision of any associated effects;
3) **Altered** data, whose precise semantics "drift" as a result of progressive enhancements, requiring corresponding updates to their effects.

This accumulating "brittleness" was particularly apparent during the development of the autonomous driving software discussed in Section III, and was one of the primary motivators of the work presented in this paper. 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 automotive industry, similar difficulties may also be encountered in the use of the Product Line[10] approach to maintain autonomous and semi-autonomous driving algorithms across a collection of related, 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 for highway merge-planning, to be reusable on all such platforms, it must be implemented in terms of the common capabilities of 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 adapted to the specific strength of each sensing technique, such as exploiting higher confidence in velocity measurements from RADAR, or the ability to detect turn signals with computer vision.

The critical observation underlying this work is that while the data that contribute to such algorithms may change from one platform to the next, the algorithms themselves remain largely the same. This work investigates the proposition that many platform-specific data will affect advanced robotic algorithms in comparatively few ways, and that identifying and encoding these likely points of variability in a dedicated "adaptation interface" would allow platform-specific enhancements, or "supplemental effects" to be maintained separately from a stable, reusable "core algorithm".

III. Supplemental Effects in Autonomous Driving Algorithms

Three autonomous driving algorithms from "Boss", Tartan Racing's winning Chevy Tahoe[11] in the DARPA Urban Challenge[12], have been thoroughly analyzed for supplemental data and effects. These algorithms were responsible for the robot's adherence to Urban Challenge traffic rules regarding safe following distance, precedence ordering at intersections, and merging into lanes of moving traffic. Their overall design, described more thoroughly in [13], emphasized adaptability and flexibility through the use of the Observer, Factory and Strategy patterns[6], encapsulating individual algorithms for traffic behaviors in separate components that could be updated or replaced individually as Boss's perception and planning capabilities evolved over time.

However, no special treatment was given to enhancing component-level adaptability, and, consequently, several behavioral algorithms became increasingly difficult to adapt, or "brittle", with respect to the robot's evolving capabilities. In particular, the perception team was continually improving the detection and tracking of other vehicles, and the system's representation of those vehicles was frequently updated as alternate sensors and modeling techniques were introduced. The final version of this so-called "Moving Obstacle" representation, shown in Figure 3, included such expected information as the position, heading, size and velocity of an obstacle that were treated as *primary* data. Beyond these, there were four *supplemental* obstacle properties that were more peculiar to Boss's final configuration:

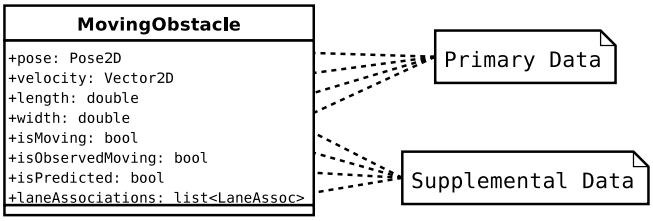

Fig. 3. The Moving Obstacle representation used for the Urban Challenge

- **Is Moving**, which was tied to the use of RADAR on Boss, indicated high confidence that the obstacle was in motion, beyond simply having non-zero velocity.
- **Is Observed-Moving**, which indicated that the obstacle's historical motion was consistent with the perception subsystem's model of "typical" Urban Challenge traffic.
- **Lane Associations**, which was a list of road lanes that the obstacle might be trying to travel in, each with a confidence value tied to similar lane-driving models in the perception subsystem.
- **Is Predicted**, which was closely tied to the robot's occlusion model, indicated that the obstacle had been extrapolated from previous sensor data, but was not instantaneously supported by sensor readings.

These properties were added and updated incrementally over many months of development, leading to frequent modifications to behavioral algorithms to accommodate new information or updated semantics, making them ideal candidates for treatment as supplemental data. Tracing the path and influence of these data yielded a total of 16 distinct supplemental effects, scattered across the implementations of three separate components:

- The **Traffic Estimator**, which identified the lead vehicle in the current travel lane and estimated the distance to and speed of that vehicle for the purposes of maintaining a safe following distance.

- The **Precedence Estimator**, which determined the precedence ordering vehicles and verified that the intersection was free of traffic before proceeding.
- The **Merge Planner**, which identified merge opportunities between other vehicles, and planned and executed merge maneuvers into adjacent lanes of travel.

The majority of the supplemental effects in these components fall into one of five categories, with the first and simplest being enhancements to relevance or false-positive culling tests. For example, the Precedence Estimator required that an obstacle have at least one strong "lane association" before treating it as candidate traffic for yield calculations. The semantics of being "associated" with a lane made the obstacle less likely to be a stalled vehicle, roadside vegetation, or other irrelevant debris, and ignoring non-associated obstacles allowed forward progress to be made in spite of those common perception artifacts.

Relatedly, the second category of supplemental effects substituted alternate thresholds into underlying calculations, often complementing the issue of culling false-positives by exploiting circumstances of increased certainty. For instance, the Traffic Estimator allowed "observed-moving" obstacles to be farther from the centerline of Boss's lane of travel before ignoring them as candidate "lead" vehicles.

The third category follows this trend of modulating the degree of "trust" placed in a candidate obstacle by affecting context-specific "conservative" estimates of the obstacle's state, such as assuming a "worst-case" speed for vehicles that are not explicitly marked as both "moving" and "observed-moving". This was the case both for the Traffic Estimator, which assumed a worst-case of "stopped", and also for the Precedence Estimator, which, for yield calculations, assumed a worst-case of the speed limit of the road.

A small number of supplemental effects fell into a fourth category, of wholesale substitution of some underlying calculation for a more efficient usage of *supplemental data*. For example, both the Precedence Estimator and Merge Planner supplanted generic, computationally-intensive geometric tests for whether an obstacle is "in" a given lane of travel with simple examination of the "lane associations" datum, once it became available. The tradeoff for this efficiency is a brittle dependency on this datum, which, if later removed, would leave these components in a non-functional state.

The fifth and final category covers secondary issues that arise from the "main" effects listed above. Examples include exposing alternate thresholds as configuration variables, and augmenting intermediate data representations to propagate *supplemental* data to the end of a long processing pipeline.

These "support" effects, along with the first two categories, of false-positive culling and the substitution of alternate weights or thresholds, are recurring themes in robotic software, suggesting that supplemental data might have similar effects in other algorithms. Following the terrain analysis example in Section I, it is conceivable that supplemental data, such as error or vegetation measurements, might cause individual points to be weighed differently in, or excluded entirely from, the embedded obstacle map calculation. This suggests a wider applicability of these ideas than this fairly narrow context of urban driving, and exploring these issues in other systems would be an excellent path of future research. In fact, some such effects have already been identified in early analysis of software from CLARAty[7], which is the origin of the `TerrainAnalysis` example above.

For the more algorithm-specific effects, such as the conservative estimation of obstacle speed, or the substitution of more efficient versions of existing functionality, it is at least plausible that there would be parallels in other robotic software, but it would be up to the judgement and experience of individual designers to identify those effects in other contexts. Part of the goal of the work described in this paper is to inform that judgement, and to equip future designers with candidate technical solutions, described in the next section, to enable them to separate these platform-specific details from the more generic, reusable elements of advanced robotic software.

IV. Technical Approaches to Modularization of Supplemental Effects

As with most software design problems, there are many possible approaches to isolating supplemental effects, such as those described above, from a given *core* robotic algorithm. The "best" solution will depend not only on a design's ability to address the problem, but also on issues such as existing development practices, team expertise, and even individual designer preference. Rather than focus on a single technical solution that may or may not meet these criteria, this work presents two distinct approaches to separating the core driving algorithms listed above from their associated supplemental effects, one based on established Object-Oriented design techniques, and one leveraging more recent developments in Aspect-Oriented methodology.

As a running example, drawn from the first category of "relevance-test" enhancements identified in the previous section, consider the pseudo-code in Listing 1, which shows a simplified segment of the original implementation of the Precedence Estimator. In this listing, the determination of whether or not a candidate obstacle will be considered for yield calculations is composed of four individual boolean tests. The first test, which represents the *core* algorithm, uses the position and size of the candidate obstacle to determine whether it occupies one or more lanes that Boss must yield to at the current intersection. Issues with detection of false positives, especially at long ranges, caused this test to be augmented by three supplemental data. An obstacle would be considered in yield calculations only if it was:

1) Marked as "moving", indicating support by one or more "solid" RADAR readings;
2) Marked as "observed-moving", indicating that its historical motion was consistent with "nominal" traffic
3) Marked as having at least one "lane association", indicating its historical motion was highly consistent with one or more real road lanes.

As shown in Listing 1, these tests were originally embedded in a large, complex method, which made them

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

Listing 1. Pseudo-code showing original, direct encoding of supplemental effects on the yield-calculation relevance test in the Precedence Estimator.

difficult to identify, understand, and update during ongoing development. These, and other such directly-encoded effects, led directly to the "calcification" of the three behavioral components discussed above, 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.

A. Object-Oriented Approach: Delegation to Strategy Classes

The first approach evaluated for modularizing these supplemental effects follows established Object-Oriented (OO) methodology by delegating segments of the core algorithm to small Strategy[6] classes that may be specialized to introduce various supplemental effects. The UML diagram in Figure 4, along with the corresponding pseudo-code in Listing 2, shows how the individual effects of the "isMoving", "isObservedMoving" and "laneAssociations" supplemental data may be isolated according to this pattern.

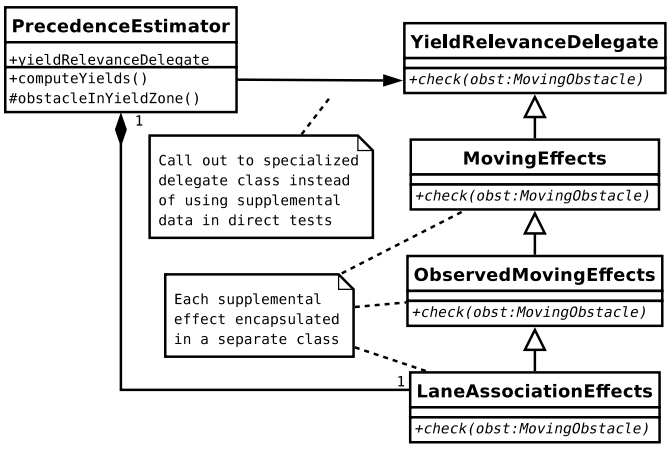

Fig. 4. Object-Oriented delegation of the yield relevance test to encapsulate supplemental effects as specialized Strategy[6] classes

The critical benefit of this alternate design is that the effects of each supplemental datum are encapsulated in individual classes, allowing them to be added, removed, or updated in isolation, as discussed above. The largest

```
// default case for YieldRelevance is "true"
class YieldRelevanceDelegate {
  vitual bool check(MovingObstacle obst) {
    return true;
  }
};
// each supplemental datum gets its own class
class MovingEffects
 : public YieldRelevanceDelegate {
   vitual bool check(MovingObstacle obst) {
     return ( obst.isMoving &&
       YieldRelevanceDelegate::check(obst) );
  }
};
// same for ObservedMoviong
class ObservedMovingEffects
 : public MovingEffects { ... };
// and for Lane Associations
class LaneAssociationEffects
 : public ObservedMovingEffects { ... };

class PrecedenceEstimator {
  // delegate are "owned" by their parent
  LaneAssociationEffects
    yieldRelevanceDelegate_;
};
void PrecedenceEstimator::computeYields() {
  // set up, same as before, but call out
  // to delegate class instead of directly
  // inspecting obstacle properties
  if(obstacleInYieldZone(obst) &&
     yieldRelevanceDelegate_.check(obst))
  {
    //  ...
  }
}
```

Listing 2. Pseudo-code showing object-oriented delegation of yield-calculation relevance test.

disadvantage of this isolation is the need to explicitly manage the dependencies between individual specialized effects, and the correspondingly long inheritance chains that follow the application of many such effects to a single delegation interface. This design also requires roughly 20 lines of code, where the "direct" encoding only took a single, albeit deeply-embedded, line for each supplemental effect. This represents a nontrivial "overhead" for encapsulating supplemental effects in this manner, which motivates exploration of other, possibly more advanced, approaches to the problem.

B. Aspect-Oriented Approach: Crosscutting Programming Interfaces

Looking beyond classical techniques, the nature of the supplemental effects described above resonates very strongly with the idea of a *crosscutting concern* as presented by the Aspect-Oriented (AO) design community[14]. That is, the effects of any one supplemental datum may be *scattered* across the implementation of one or more *core* algorithms, and they often *tangle* with effects of other supplemental data by affecting the same places in those algorithms, such as in the "yield relevance" example presented above.

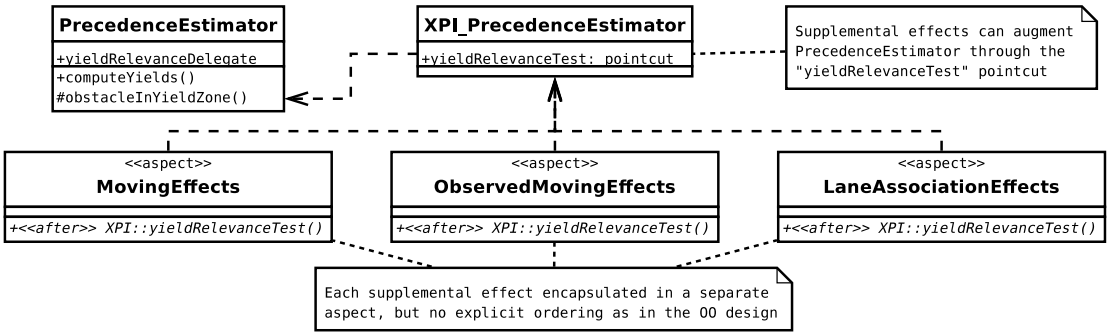

Fig. 5. Aspect-Oriented exposure of adaptability through a Crosscutting Programming Interface (XPI), and binding supplemental effects as "after" advice through the XPI

AO methodology is aimed at a more natural representation of such *crosscutting concerns* than is possible using traditional OO approaches, making AO techniques an attractive solution to the problem of encapsulating the effects of supplemental data as described above. In particular, the AO design community has recently proposed the idea of a "Crosscutting Programming Interface"[15], or "XPI", which complements traditional functional interfaces with an enumeration of the ways that a given component may be augmented by AO "advice" introductions. Although originally conceived in terms of more benign effects on a software system, such as debug tracing, transaction logging, or thread synchronization concerns, an XPI is highly consistent with the idea of an "adaptation interface" proposed above. As shown in Figure 5, and the corresponding Listing 3, an XPI may be used to allow similar encapsulation of supplemental effects as in the OO design without the same degree of interdependencies or raw overhead.

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. This leads to a much more concise description of these effects, averaging fewer than 10 lines of code each, where the OO design takes closer to 20 lines per supplemental effect. The drawback to this approach is that AO techniques are still relatively new, which means that tool support is somewhat limited and prototypical, and the overall methodology will be unfamiliar to most developers, both of which pose significant risks to the success of any project.

```
void PrecedenceEstimator::computeYields() {
  // set up, same as before, but no extra
  // calls or conditions in relevance test
  if(obstacleInYieldZone(obst))
  {
    //  ...
  }
  // ...
}
// 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 MovingEffects {
  // augment yield relevance test
  // require obst->isMoving
  advice XPI_PrecedenceEstimator::
    yieldRelevanceTest(obst) : after()
  {
    // tjp (The Join Point) allows return
    // value to be manipulated by advice
    *(tjp->result()) &= obst.isMoving;
  }
}
// .. same for observed-moving, lane assoc.
// no explicit composition as in OO design
```

Listing 3. Pseudo-code showing application of supplemental yield relevance effects through an XPI

C. Analytical Results and Discussion

All 16 supplemental effects in the Urban Challenge software system were refactored following each of the two designs described above, and the resulting artifacts have been analyzed using two different software metrics that quantify:

1) The degree of "concern diffusion"[16] in a given system, which is a proxy for "understandability";
2) The "net option value"[17] provided by a design, which models how well it accommodates changes by modular substitution, instead of direct modification.

A complete presentation of these results requires detailed discussion of individual supplemental effects that is beyond the scope of this paper. Instead, Table I summarizes the average- and best-case changes in "diffusion" and "option value" for supplemental effects in each of the OO and AO designs, as compared to the original implementation.

These results indicate that both of the detailed design techniques can enhance a component's adaptability to platform-specific details, but that the AO design may be better-suited to this problem. This is consistent with the examples in the previous section, where the AO technique was both more concise at the source-code level, and created fewer dependencies between components at the design level. Still, the AO design represents a nontrivial technical risk, and there

Metric →	Concern Diffusion		Net Option Value	
Design ↓	Average	Best	Average	Best
OO Delegates	-27%	-42%	+51%	+67%
AO XPI's	-100%	-100%	+58%	+80%

TABLE I

HIGH-LEVEL SUMMARY OF ANALYTICAL RESULTS. BETTER DESIGNS HAVE LESS "DIFFUSION" AND MORE "OPTION VALUE".

are still clear benefits to be had through the more traditional OO design, so which approach is "best", or whether any such advanced design is warranted at all, is ultimately up to the judgement of individual designers.

Even in the event that these advanced designs represent too much risk, or too much additional work, the refactoring and analysis described above also suggests several ways that the proposed *primary* vs. *supplemental* methodology can guide much simpler design processes. For example, the categories of supplemental effects described in Section III can be used to guide the decomposition of large functions into smaller subsidiary functions. Something as simple as isolating the elements of Listing 1 that depend on supplemental data into a separate method, such as `obstacleIsRelevantForYields()`, would make it easier to identify existing supplemental effects in the larger `PrecedenceEstimator` class. Taken one step further, dedicating such methods for other likely points of variability, even if the "base case" is simply to return "true", could pay significant dividends to future adaptation by highlighting the places where platform-specific effects might be applied. Moreover, such a method-level decomposition would make it easier to apply more advanced design techniques at a later time, promoting their incorporation on an as-needed basis.

V. SUMMARY

This paper has presented a design methodology aimed at enhancing the adaptability and reusability of advanced robotic software components based on the isolation of the platform-invariant *core* algorithm from the *supplemental* effects that enhance that algorithm to exploit the special capabilities of any one robot. The division between the two is framed in terms of the *primary* data that all robots must provide to a given *core* algorithm, and the remaining platform-specific *supplemental* data, which will vary from one robot to the next.

Five categories of *supplemental* effects have been identified in autonomous driving algorithms, including:

- Explicit policies for determining whether a candidate obstacle is "relevant" to the present context;
- Specific thresholds for determining the scope, or "maximum range" of the present context;
- Conservative estimations of obstacle state, such as assuming a context-specific "worst-case" speed;
- Substitution of generic, resource-intensive calculations with more efficient usage of platform-specific data;
- Propagation of supplemental data through intermediate data representations in long processing "pipelines".

Two advanced design techniques, one aspect-oriented and one object-oriented, have been proposed to separate these effects from their underlying core algorithms. High-level results from applying these designs to existing software indicate that they can increase the understandability and adaptability of the corresponding software artifacts, supporting their consideration for future design efforts.

Even though this work is framed in the context of a single software system, the categories of supplemental effects listed above are likely have analogues in other advanced robotic software, suggesting a broader applicability of the issues outlined in this paper. This encourages future developers to consider the distinction between *primary* and *supplemental* data, and a corresponding division of *core* algorithms from *supplemental* effects to enhance the adaptability and reusability of advanced robotic software.

REFERENCES

[1] M. Montemerlo, N. Roy, and S. 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.

[2] T. H. Collet, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proceedings of the Australasian Conference on Robotics and Automation (ACRA), Sydney, Australia*, 2005.

[3] I. Nesnas, *et al.*, "CLARAty: An architecture for reusable robotic software," in *Proceedings of SPIE*, 2003.

[4] M. Quigley, *et al.*, "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.

[5] C. Cote, *et al.*, "Robotic software integration using MARIE," *International Journal of Advanced Robotic Systems*, vol. 3, pp. 55–60, 2006.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[7] I. A. Nesnas, *et al.*, "CLARAty: Challenges and steps toward reusable robotic software," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 023–030, 2006.

[8] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Essex, UK, UK: Addison-Wesley Longman Ltd., 1999.

[9] A. Cowley, L. Chaimowicz, and C. J. Taylor, "Design minimalism in robotics programming," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 31–36, November 2008.

[10] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 2003.

[11] C. Urmson, *et al.*, "Autonomous Driving in Urban Environments: Boss and the DARPA Urban Challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[12] Defense Advanced Research Projects Agency (DARPA), "Urban challenge website," July 2007, http://www.darpa.mil/grandchallenge.

[13] C. R. Baker and J. 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*, vol. 16, no. 1, pp. 78–87, 2009.

[14] G. Kiczales, *et al.*, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[15] W. G. Griswold, *et al.*, "Modular software design with crosscutting interfaces," *IEEE Softw.*, vol. 23, no. 1, pp. 51–60, 2006.

[16] A. F. Garcia, *et al.*, "Modularizing design patterns with aspects: A quantitative study." *Transactions on Aspect-Oriented Software Development I*, vol. 3880/2006, pp. 36–74, 2006.

[17] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. 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*. New York, NY, USA: ACM, 2001, pp. 99–108.