6 Evaluation

The following sections will cover the evaluation of ASMS, as well as examine some of the code-related elements of each exemplar that will provide additional context to the evaluation. We will (1) analyze the technical details of each exemplar's implementations using Understand, (2) present how ASMS may be applied to identified adaptive strategies, (3) explain the experimental setup behind how these metrics will be evaluated, and (4) discuss the derived results.

6.1 Case Studies in Understand

In order to gain a better awareness of how these exemplars function with regards to their implementations, this section will detail a high level overview of the code-related properties of each case study using Understand.

Table 9 and 10 provides details of the five exemplars with respect to their code volume. Table 9 shows their structural overview by specifying the programming language used, and the number of files, classes and functions within each codebase. Table 10 presents slightly more in depth information by showing the number of lines of code and the comment to code ratio. It also shows the number of executable and declarative statements.

	Programming Language	Number of Files	Number of Classes	Number of Functions
Platooning LEGOs	Python	4	0	0
ATRP	Java	56	71	819
Dragonfly	Java	43	43	278
UNDERSEA	Java, C++	81	78	526
TRAPP	Python	31	13	79

Table 9: Table showing the programming language and number of files, classes and functions of each examined exemplar

From Table 9, one can observe that ATRP and UNDERSEA have the most expansive codebases out of all the exemplars. By contrast, the Platooning LEGOs only has 4 files containing all the code, which represent the possible adaptive strategies at the system's disposal.

	Number of Lines of Code	Comment to Code Ratio	Number of Executable Statements	Number of Declarative Statements
Platooning LEGOs	911	0.54	438	192
ATRP	5,510	1.12	1,069	950
Dragonfly	10,888	0.18	2,987	2,445
UNDERSEA	7,905	0.28	2,103	1,573
TRAPP	30,782	0.01	633	409

Table 10: Table showing the number of lines of code and the comment to code ratio, as well as the number of executable and declarative statements of each exemplar.

Going one level more in depth, one can observe from Table 10 that the Dragonfly exemplar contains the most amount of runnable code since it possesses the largest number of executable and declarative statements. TRAPP has the most amount of lines of code, but a large amount of this is dedicated to storing the data related to the maps that the cars drive on. Similar to Table 9, the Platooning LEGOs also has the smallest codebase in regards to the volume of its code.

6.2 Collecting the Results from ASMS

The following subsection will explain how the adaptive strategies of each exemplar have been identified, as well as how ASMS may be applied to each of them.

6.2.1 Identifying the Adaptive Strategies

The adaptive strategies of each exemplar have been identified through their accompanying paper. The majority of the strategies are explicitly mentioned and given explanations on how they function. These can then be found and analyzed in Understand.

It is important to note that for some exemplars, the word *adaptive strategy* is not always used to describe the components that a SAS uses to achieve self-adaptation. For example, in the UNDERSEA exemplar, the authors use the term *controllers* as a substitute for adaptive strategies, but these are functionally very similar to the strategies in other exemplars.

6.2.2 Applying ASMS

Once the adaptive strategies of an exemplar have been identified, Understand will be able to apply ASMS. The first step is to group them together into an architecture, which is a feature that allows a user in Understand to organize pieces of code together. The architecture consists of a set of entities (see Section 5.1.3) that correspond to an adaptive strategy in the exemplar. After this step, ASMS can then be run on the architecture as an IReport (see Section 5.1.1), which is shown in Figure 5 using the TRAPP adaptive strategies as an example.

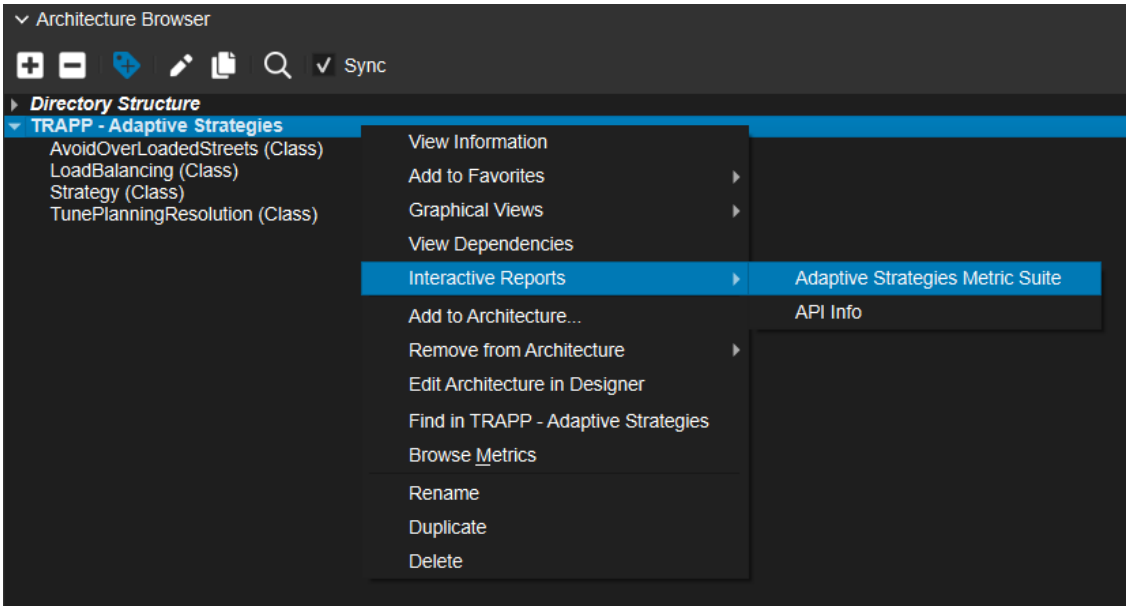

Figure 5: Screenshot showing how to run the ASMS as an IReport, using the adaptive strategies of the TRAPP exemplar as an example.

After running the ASMS IReport, Understand should produce a UI box in the bottom right that displays all the results from the metrics. An example of this is shown in Figure 6, also using TRAPP.

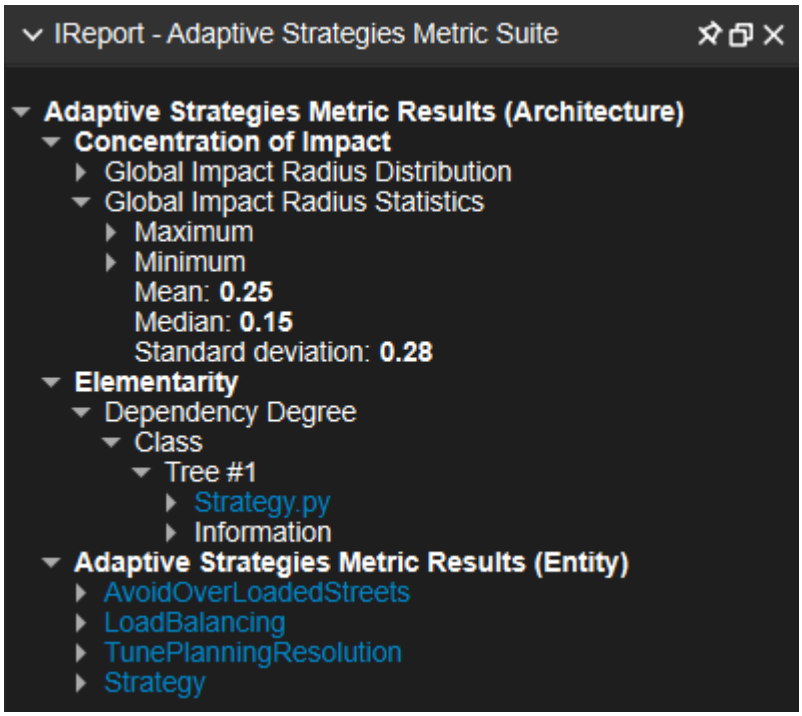

Figure 6: Screenshot showing the results after ASMS has been run on the architecture shown in Figure 5. By default, only the GIRS are immediately shown after ASMS displays the results.

After immediately running ASMS on a given architecture, the GIRS metric will be shown by default. The other dropdown trees can be expanded to see the results from the other metrics. The reason why the GIRS metric dropdown tree is collapsed by default is because it does not take up a significant amount of space in the UI box, which prevents a new user from being exposed to too much information simultaneously.

The following subsection will show the setup used for the evaluation of two exemplars (TRAPP and ATRP), as well as showing their results using graphs and diagrams. These results are collected based on the displayed outputs from ASMS, as shown in Figure 6.

6.3 Experimental Setup

Two exemplars have been selected that the metrics will be executed on. These are TRAPP and ATRP. The metrics will run on the adaptive strategies of each exemplar and present the results. Table 11 shows the general details of the adaptive strategies involved.

Exemplar	Programming Language	Dataset to be evaluated on (Adaptive Strategy)	Input Type
TRAPP	Python	`Strategy`	Class
		`AvoidOverLoadedStreets`	
		`LoadBalancing`	
		`TunePlanningResolution`	
ATRP	Java	`AbstractRoutingAlgorithm`	Class
		`LookaheadShortestPathRoutingAlgorithm`	
		`QLearningRoutingAlgorithm`	
		`TrafficLookaheadRoutingAlgorithm`	
		`AlwaysRecomputeRoutingAlgorithm`	
		`AdaptiveRoutingAlgorithm`	

Table 11: Table showing the list of adaptive strategies that ASMS will run on. For each strategy, its corresponding exemplar, programming language and input type is also noted.

The adaptive strategies associated with TRAPP represent the options that cars have in navigating the city to reach their destination. Likewise, the adaptive strategies associated with ATRP represent the different routing algorithms that the vehicles use to navigate the city. These incorporate self-adaptation to tell the managed system on how to respond to incoming environmental and internal changes during runtime, such as traffic accidents or congestion.

The justification for why these particular exemplars have been chosen to evaluate on is because each approaches a SAS with the a similar goal, but with different methods. Firstly, the programming language of choice for ATRP is Java, while that of TRAPP is Python. This is able to show that the metrics design and plugin are capable of executing on multiple types of implementation languages.

Secondly, these exemplars both have multiple adaptive strategies, which means that the application of metrics such as the Concentration of Impact and Elementarity are possible.

For each of the adaptive strategies in their respective architecture (as shown in Figure 5), each of the implemented metrics presented in Section 4.2 will be run. These include the following:

1. Locality
   (a) Impact Radius
2. Concentration of Impact
   (a) Global Impact Radius Distribution
   (b) Global Impact Radius Statistics
3. Elementarity
   (a) Dependency Degree
4. Maintainabiltiy
   (a) Adaptive Complexity
   (b) Adaptive Modifiability

All subtypes of the metrics listed above will be tested on the adaptive strategies in Table 11. The following section will explain the experiment behind each metric as well as discuss the results.

6.4 Results

6.4.1 Locality Metrics Results

This experiment applies the Local and Global Impact Radius to ATRP (Figure 7) and TRAPP (Figure 8) using the methods discussed in Section 4.2.1. Carrying out this experiment can show how the metric is able to determine which adaptive strategies in a system can have the most impact within a system. For each of the graphs presented below, we see the number of reachable executable statements on the y-axis, and the names of the adaptive strategy on the x-axis. Furthermore, each bar is color coded to indicate if it is the Local Impact Radius or the Global Impact Radius.

Figure 7 shows the graph corresponding to the Local and Global Impact Radius of each adaptive strategy in ATRP. From this data, it is clear that the `QLearningRoutingAlgorithm` is able to reach the most executable statements, followed by the `LookaheadShortestPathRoutingAlgorithm`. This tells us that both of these have the greatest potential to affect change in the system.

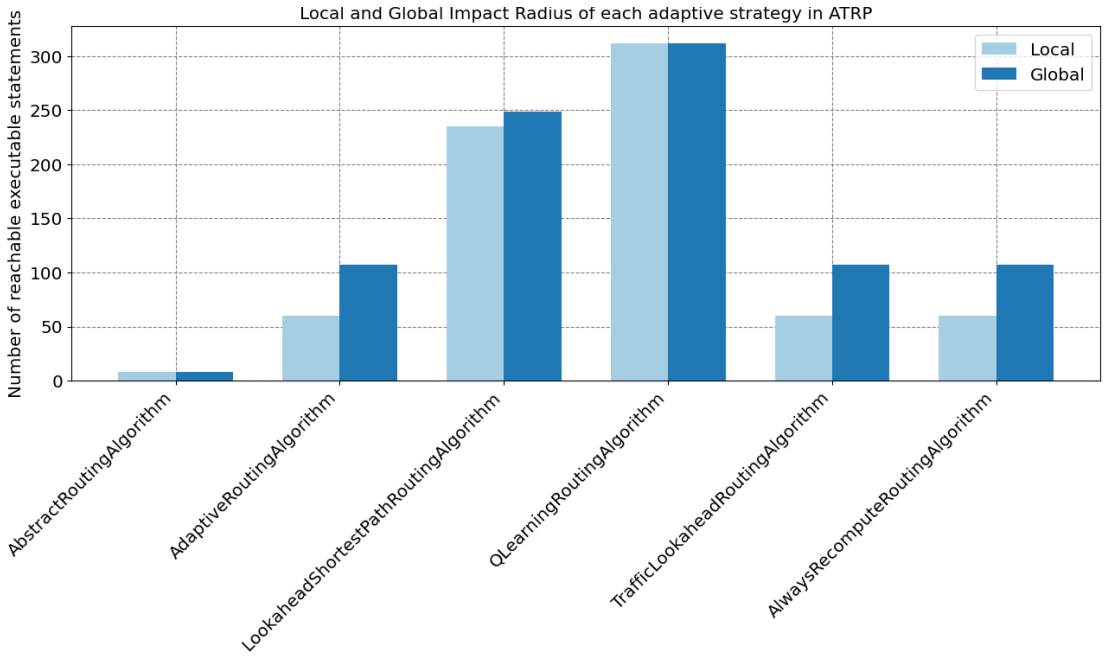

Figure 7: Double bar chart showing the Local and Global Impact Radius of each adaptive strategy in ATRP, with the number of reachable executable statements on the y-axis and the adaptive strategies on the x-axis.

Other outstanding properties may be observed in the discrepancies between the LIR and GIR of an adaptive strategy. For example, the `AdaptiveRoutingAlgorithm` (among others) has a clear difference between the number of executable statements that may be reached locally versus globally. This indicates that the strategy has a much greater impact on the more distant parts of the system. By contrast, the `QLearningRoutingAlgorithm` has no difference between its LIR and GIR, which means its impact is restricted to the parts of the SAS that are closely associated to it.

Figure 8 shows the LIR and GIR when applied to the TRAPP implementation. In this case, the metric shows that the impact radius of each adaptive strategy is much more one-sided, with the `TunePlanningResolution` strategy being able to reach the greatest number of executable statements in the system. This tells us that the system has to spend more resources on handling that particular strategy in contrast to the others. Alternatively, strategies such as `Strategy` and `LoadBalancing` have less impact.

Strikingly, all strategies also have little difference between their LIR and GIR, which also indicate that they limit their impact to a small, restricted subset of the SAS they operate on.

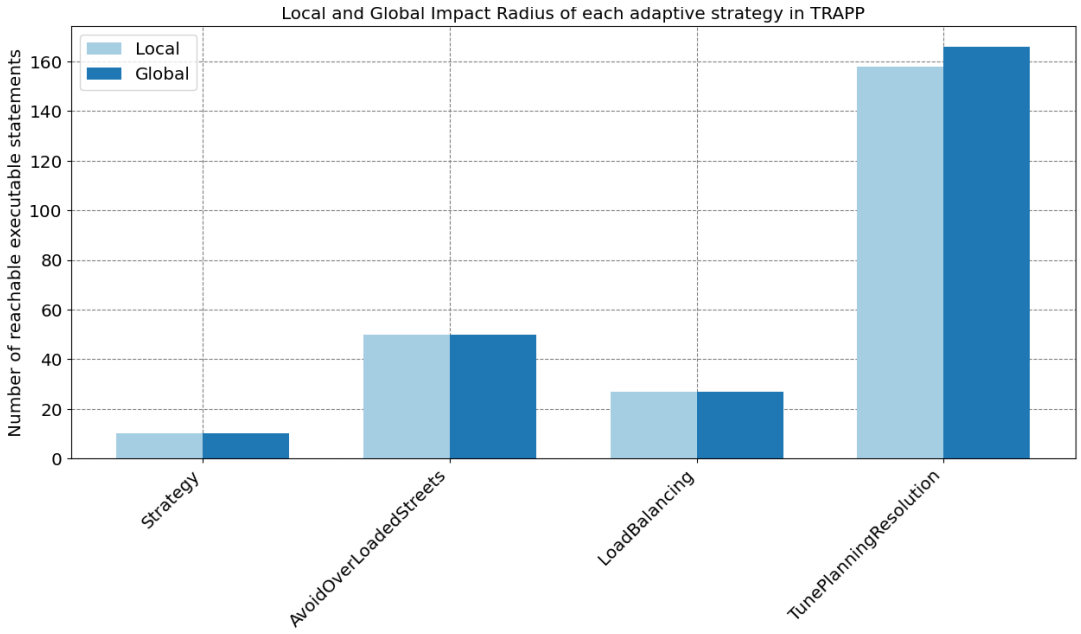

Figure 8: Double bar chart showing the Local and Global Impact Radius of each adaptive strategy in TRAPP, with the number of reachable executable statements on the y-axis and the adaptive strategies on the x-axis.

Limitations and Future Work. A possible limitation that can be observed in the Impact Radius metric is in the decision to count the number of reachable executable statements to measure how much impact an adaptive strategy has on a SAS. Using this method, the metric is not able to account for changes in the SAS related to when the value of a highly important variable is changed. If the behaviour of the system is greatly dependent on the value of this one particular variable, then the metric would not be able to detect this, as it would simply count it as a single executable statement.

In order to remedy this problem and get a more accurate reading of an adaptive strategy's impact, future work could involve counting how many references a variable has. A reference to a variable can be defined as whenever the variable is declared, or when a variable's value is read or written to. The more references the given variable has across the system, the greater an indication it could give that the system is more dependent on the value of that particular variable. Therefore, the Impact Radius could take into account two different facets of a piece of code: how many executable statements it can locally and globally reach as well as how many references its constituent variables have.

6.4.2 Concentration of Impact Metrics Results

This experiment applies the Global Impact Radius Distribution (GIRD) metric to ATRP (Figure 9) and TRAPP (Figure 10) using the methods discussed in Section 4.2.2. It defines the set of each adaptive strategy's Relative Global Impact Radius (RGIR) and can be used to compare which strategies have the most impact within a system. For each of the graphs presented below, we see a pie chart showing the RGIR of each strategy from the computed GIRD metric.

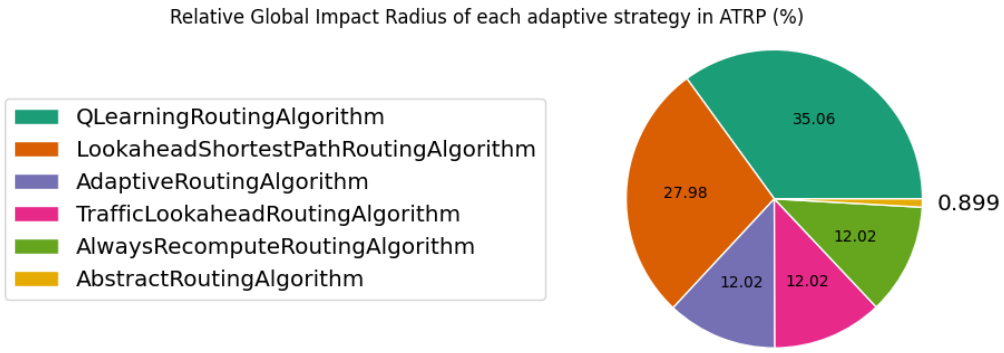

Figure 9: Pie chart showing the Relative Global Impact Radius of each adaptive strategy in ATRP, expressed as a percentage.

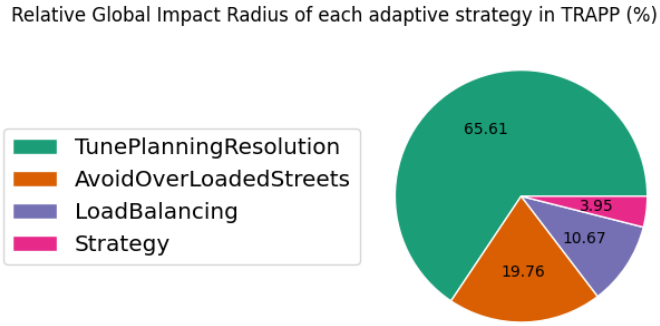

Figure 10: Pie chart showing the Relative Global Impact Radius of each adaptive strategy in TRAPP, expressed as a percentage.

Figure 9 and 10 show the RGIR of the ATRP and TRAPP implementations. From there, we can see how the RGIR is distributed among the adaptive strategies that each exemplar uses. In contrast to the Locality metrics, this approach can provide an alternate view on

how much impact an adaptive strategy has on a system relative to others. However, many of the same observations made in the section above apply here as well.

The usefulness of the GIRD metric can be exemplified in how we can use it to calculate statistics, which is provided by the Global Impact Radius Statistics (GIRS) metric. This includes statistics that show the adaptive strategy with the maximum and minimum RGIR, as well as the mean, median and standard deviation of the RGIR of all adaptive strategies. Table 12 shows the outputs for this metric based on each exemplar.

	ATRP	TRAPP
Maximum	`QLearningRoutingAlgorithm`	`TunePlanningResolution`
Minimum	`AbstractRoutingAlgorithm`	`Strategy`
Mean	0.17	0.25
Median	0.12	0.15
Standard Deviation	0.12	0.28

Table 12: Table showing the Global Impact Radius Statistics of ATRP and TRAPP, including the Maximum, Minimum, Mean, Median and Standard Deviation subtypes

From this table, we can see how GIRS provides some unique interpretations to the GIRD's data. Firstly, it can concretely give the adaptive strategy that has the maximum or minimum RGIR through the first two statistics.

Secondly, the Mean and Median subtypes can show around which RGIR values the adaptive strategies tend to exist. The Mean can show the average RGIR and the Median can show the middle value RGIR of the strategies. The difference between these two values is also particularly useful in getting an idea of how the strategies are distributed in regards to their RGIR, since the Mean may be greatly affected by extreme values, while the Median is not.

Thirdly, it can also show how spread out the RGIRs are of the adaptive strategies. For example, from Table 12 we can see that the TRAPP exemplar has adaptive strategies that have RGIRs that are more varied since it has a higher standard deviation. This is a useful piece of information because in systems where this statistic is high, it points to the existence of one or more adaptive strategies that can reach significantly more executable statements relative to other strategies.

Limitations and Future Work. In regards to limitations, the current repertoire of statistics made available by the GIRS metric can be further expanded upon. This would

ensure a wider variety of information, as the more statistics that are present in the metric, the more insights it can provide into the impact of the adaptive strategies within a SAS. Possible candidates to add could be the range and interquartile range, which can further highlight how impact is spread out among strategies.

In addition to this, adding features that are capable of dynamically rendering graphs can also be a useful extension to the plugin. This would make it easier to intuitively understand the data since it has a visual element to it. However, since the Understand API does not currently allow the use of imported libraries, the plugin may need to be completely reprogrammed for another tool in order for this addition to work.

6.4.3 Elementarity Metrics Results

This experiment applies the Dependency Degree metric to ATRP (Figure 11) and TRAPP (Figure 13) using the methods discussed in Section 4.2.3. Through this metric, it is possible to determine how dependent (and in what ways) adaptive strategies are on one another. For each of the graphs presented below, the x-axis represents the name of the adaptive strategy, whereas the y-axis presents the number of dependency relations that adaptive strategy has with another in the architecture. It is also color-coded based on the subtype that is being measured, with those being *Depends on* and *Depended on by*.

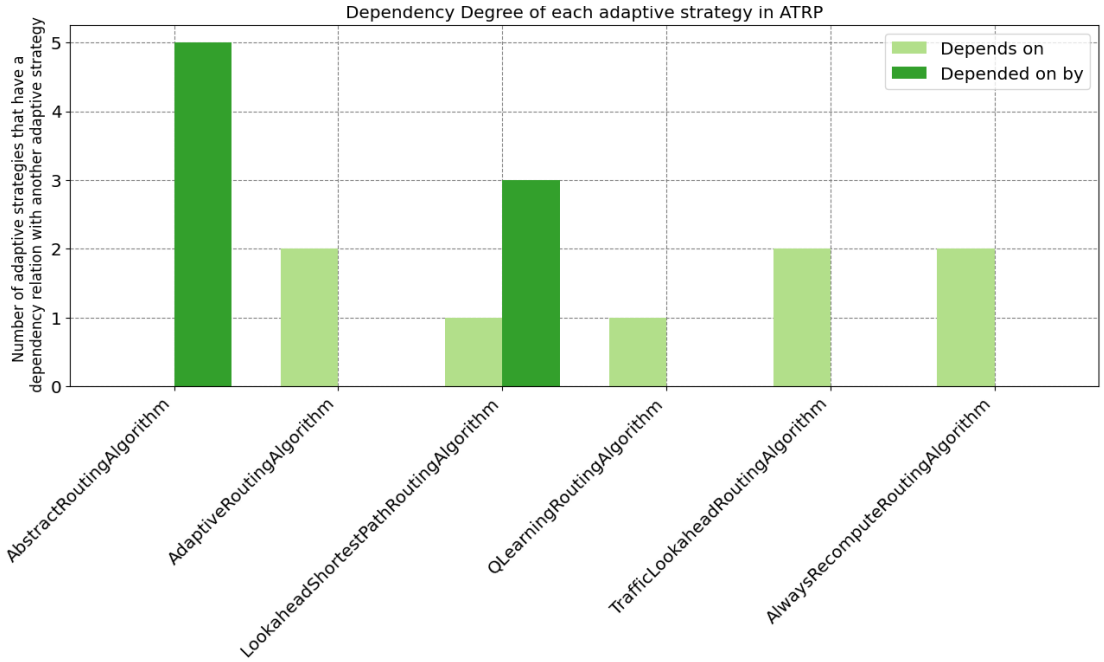

Figure 11: Double bar chart showing the *Depends on* and *Depended on by* subtypes when applied to ATRP, with the number of dependency relations a given adaptive strategy has on the y-axis and the adaptive strategies themselves on the x-axis.

The above graph shows the Dependency Degree metric results for the ATRP implementation. From these results, we can see that the `AbstractRoutingAlgorithm` is depended on by the most amount of strategies, while several others are depended on by zero. On the other hand, a handful of strategies are tied for depending on the most amount of other strategies.

There are a number of properties that are implied by this data. Firstly, since the `AbstractRoutingAlgorithm` depends on zero strategies, while being depended on by 5 (i.e., all other strategies), it indicates that it has influence over all other strategies in the architecture. Since each strategy in this architecture is a class, this tells us that each one extends functionality from the `AbstractRoutingAlgorithm` through inheritance. Secondly, every strategy that is depended on by zero others, but does depends on at least one, constitutes a strategy that is not inherited from.

These two notions are reflected in Figure 12, which shows the results from the *Tree* subtype of the Dependency Degree metric. The adaptive strategy that is depended on by all others, but does not depend on any, forms the root of the tree. Likewise, the adaptive strategy that is not depended on by any, but does depend on at least one forms a leaf in the tree.

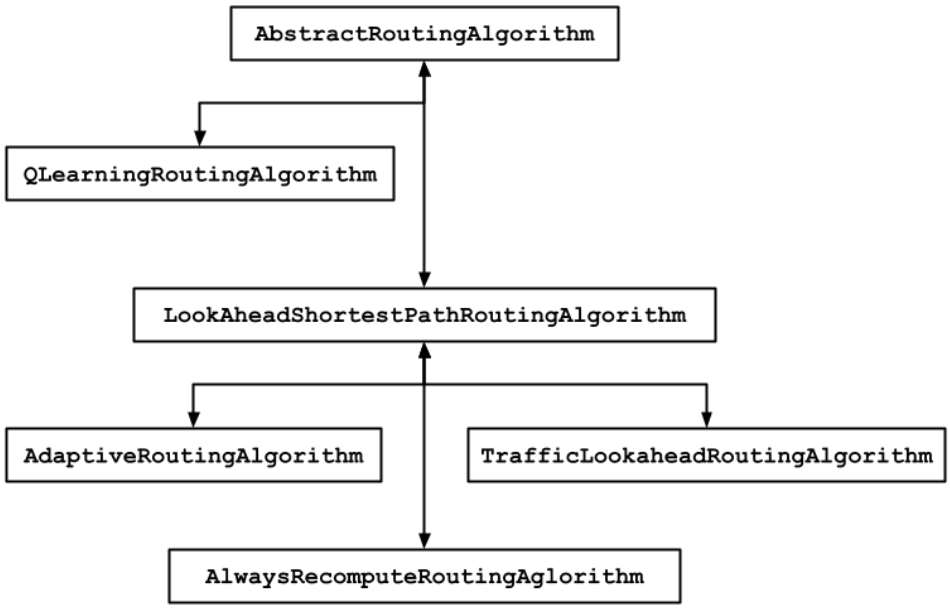

Figure 12: Diagram showing the Dependency Degree Tree of the ATRP adaptive strategies, with parent nodes representing the strategy a given strategy depends and child nodes representing the strategies a given strategy are depended on by.

Leading on from this, Figure 13 shows the results of the Dependency Degree when applied to the TRAPP implementation. According to the output of the metric, the `Strategy` class is depended on by all other strategies, while all other strategies depend on it.

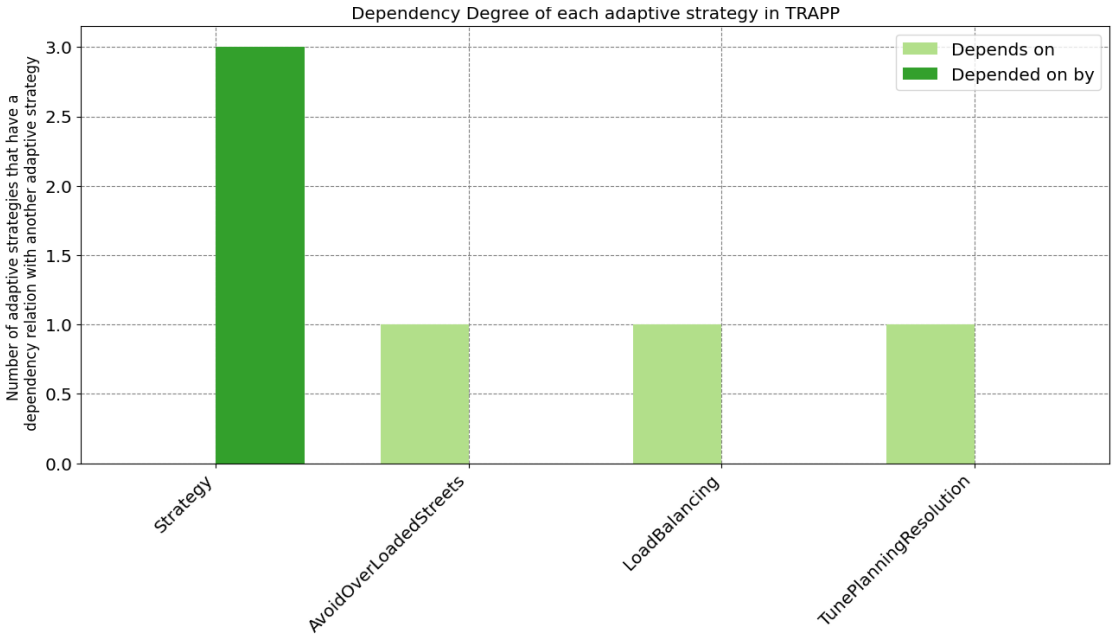

Figure 13: Double bar chart showing the *Depends on* and *Depended on by* subtypes when applied to TRAPP, with the number of dependency relations a given adaptive strategy has on the y-axis and the adaptive strategies themselves on the x-axis.

Therefore, the corresponding *Dependency Degree Tree* may be drawn as shown in Figure 14. For each of the adaptive strategies in Figure 14, the `Strategy` class forms the root of the tree since it is depended on by all others and depends on none. By contrast, the `AvoidOverLoadedStreets`, `LoadBalancing` and `TunePlanningResolution` form the leaves of the tree because they all depend on `Strategy`, but are not depended on by any.

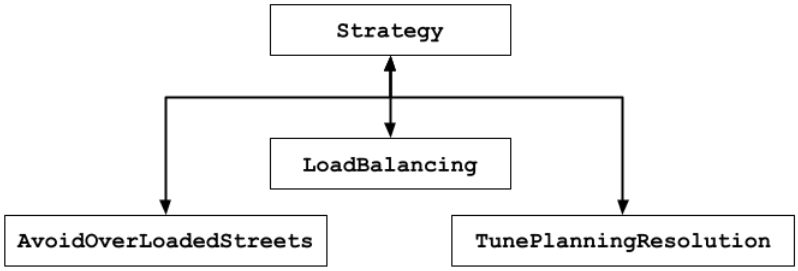

Figure 14: Diagram showing the Dependency Degree Tree of the TRAPP adaptive strategies, with parent nodes representing the strategy a given strategy depends and child nodes representing the strategies a given strategy are depended on by.

Limitations and Future Work. One of the identified limitations can be observed when the metric is run on an architecture of methods. A dependency relation, as one may recall from Section 4.2.3, does not only include when one adaptive strategy is dependent on the functionality of another (as is the case in the ATRP strategies), but also when one strategy is dependent on the execution of another. In the case of a method, this can be seen when one adaptive strategy calls another, thereby creating a connection between the two in the former strategy's call graph.

However, a dependency may also be created when one adaptive strategy is locked behind a conditional involving another. This is best illustrated using the code snippet present in Listing 2.

```
1
2     boolean strategy_1_performed = perform_strategy_1(...);
3     if (strategy_1_performed) {
4         boolean strategy_2_performed = perform_strategy_2(...);
5         if (strategy_2_performed) {
6             boolean strategy_3_performed = perform_strategy_3(...);
7             if (strategy_3_performed) {
8                 ...
9             }
10        }
11    }
12
```

Listing 2: Code snippet showing an example where the Dependency Degree metric fails to identify dependencies based on the conditional execution between adaptive strategies.

According to the code written above, the `strategy_1` strategy is depended on by the execution of `strategy_2`, which itself is depended on by the execution of `strategy_3`. In

this example, a given strategy is locked behind a conditional that can only be accessed when another strategy is performed first. The current implementation of ASMS is not able to detect a dependency when adaptive strategies are performed in this manner.

Insofar as future work is concerned, Understand could alleviate this problem since its API does allow access to the control flow graph of a method or function. An algorithm would still need to be developed which takes advantage of this feature and that can reliably compute the dependency relations in such a way that the problem in Listing 2 is mitigated.

6.4.4 Maintainability Metrics Results

6.4.4.1 Adaptive Complexity Results. This experiment applies the Adaptive Complexity metric to ATRP (Figure 15) and TRAPP (Figure 16) using the methods discussed in Section 4.2.4. This metric is focused on calculating the complexity of each adaptive strategy. For each of the graphs presented below, the y-axis represents the number of linearly independent paths through the control flow graphs involved in an adaptive strategy, while the x-axis represents the names of each strategy. It is also color-coded based on the subtypes of the metric, which are the Cyclomatic Complexity (CC) and Strict Cyclomatic Complexity (SCC).

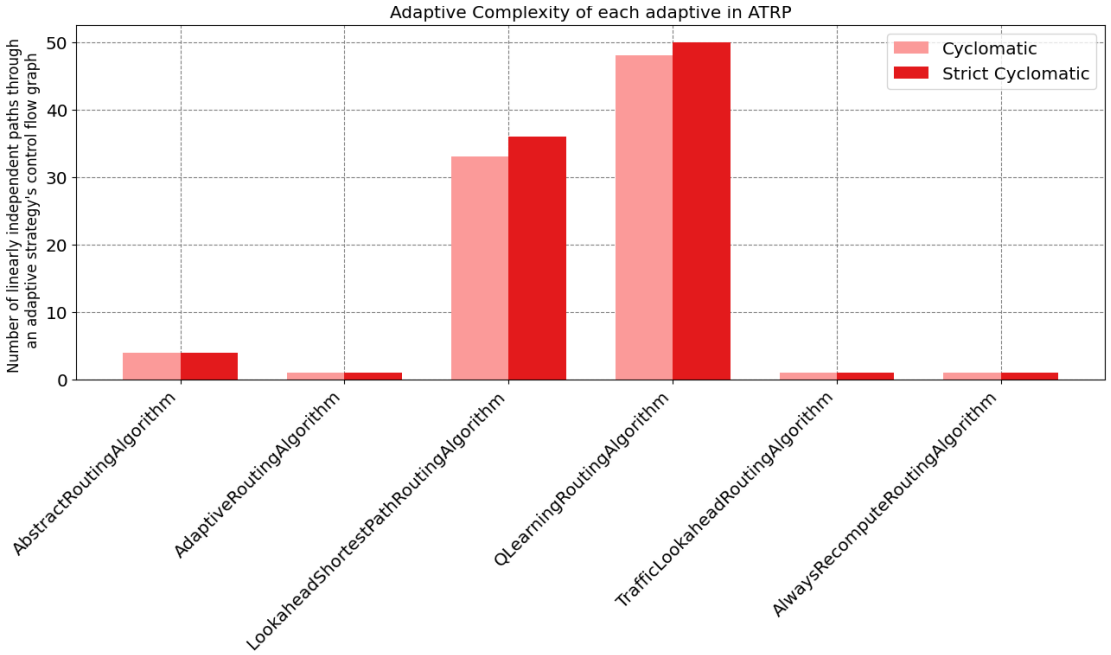

Figure 15: Double bar chart showing the Cyclomatic and Strict Cyclomatic Complexities of the ATRP adaptive strategies, with the y-axis representing the number of linearly independent paths through an adaptive strategy's control flow graph and on the x-axis the names of the adaptive strategies.

Figure 15 shows the results pertaining to the complexity of the ATRP adaptive strategies. Based on these results, it is clear that the `QLearningRoutingAlgorithm` and `LookaheadShortestPathRoutingAlgorithm` are the most complex out of all the adaptive strategies. One indication that this observation provides is that these strategies will require significantly more effort to test appropriately. By contrast, all other strategies have a CC and SCC of less than 5, which will be less costly to test.

In addition to this, Figure 16 shows the results from the adaptive strategies used in TRAPP. According to these results, the `TunePlanningResolution` adaptive strategy is the most complex by a significant margin. Similar to ATRP, there is one highly intricate strategy which is accompanied by several less intricate ones.

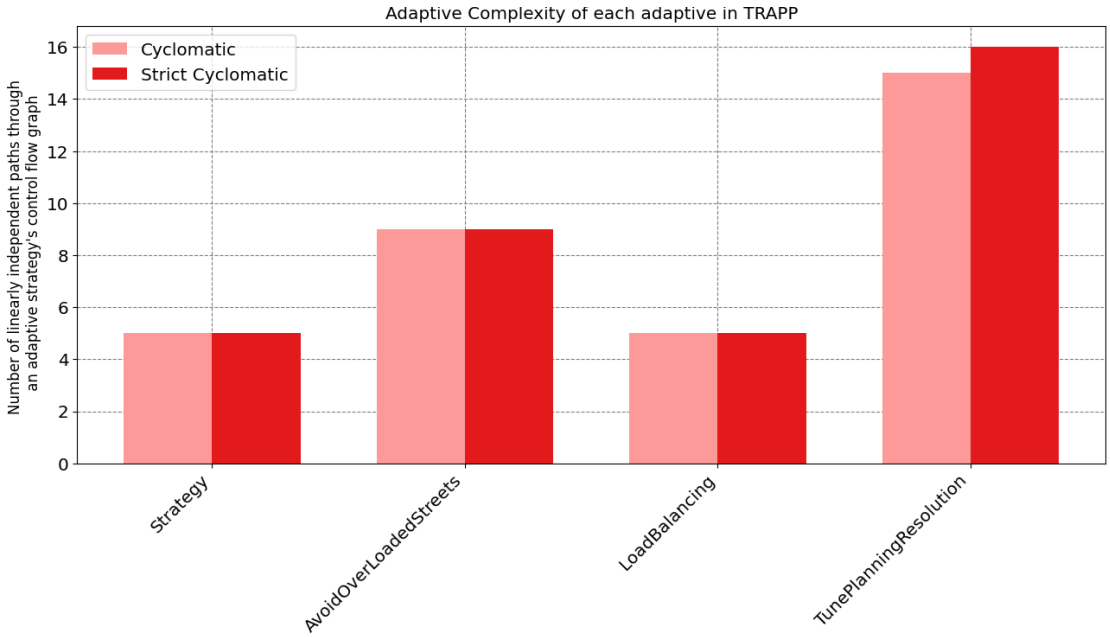

Figure 16: Double bar chart showing the Cyclomatic and Strict Cyclomatic Complexities of the TRAPP adaptive strategies, with the y-axis representing the number of linearly independent paths through an adaptive strategy's control flow graph and on the x-axis the names of the adaptive strategies.

In light of these results, there is a compelling finding that should be noted. On closer inspection, it can be observed when comparing the results of this metric to that of the Local and Global Impact Radius (Figure 15 compared to 7, and Figure 16 compared to 8). Putting both these results side by side, we can see that there are similarities and overlaps between them. This can be explained in part due to the fact that more complex classes and functions have a greater chance to contain executable statements, since there will be more branching paths that may be followed.

Limitations and Future Work. Computing only the Cyclomatic Complexity of the adaptive strategies may not be able to provide a complete picture into how complex the strategies are. The Adaptive Complexity metric is limited in that it does not take into account factors such code volume or data flow, which also play a role in defining the intricacies of a piece of code [15]. It only focuses on the control-flow as a means to measure complexity.

As a result, future work may include adding more subtypes that measure the complexity of a strategy in different ways. For example, the metric may benefit from the addition of *Halstead's Measures of Complexity*, which considers factors such how program difficulty and effort based on the number of operands and operators in a piece of code [15].

In addition to this, complexity may also be further measured from an object-oriented point of view. If adaptive strategies tend to present themselves in the form of a class, it may be conducive to measure how the interactions between strategies contribute to their complexity. This could include examining factors such as the role of inheritance or how data is handled between classes [15].

6.4.4.2 Adaptive Modifiability Results. This experiment involves applying the Adaptive Modifiability metric of the ATRP and TRAPP exemplar. In regards to the Coupling subtype, the graphs presented below (Figure 17 and 18) shows the Coupling of each adaptive strategy to other modules in ATRP and TRAPP. On the x-axis, we see the names of the adaptive strategies and the y-axis, the number of classes they are coupled to.

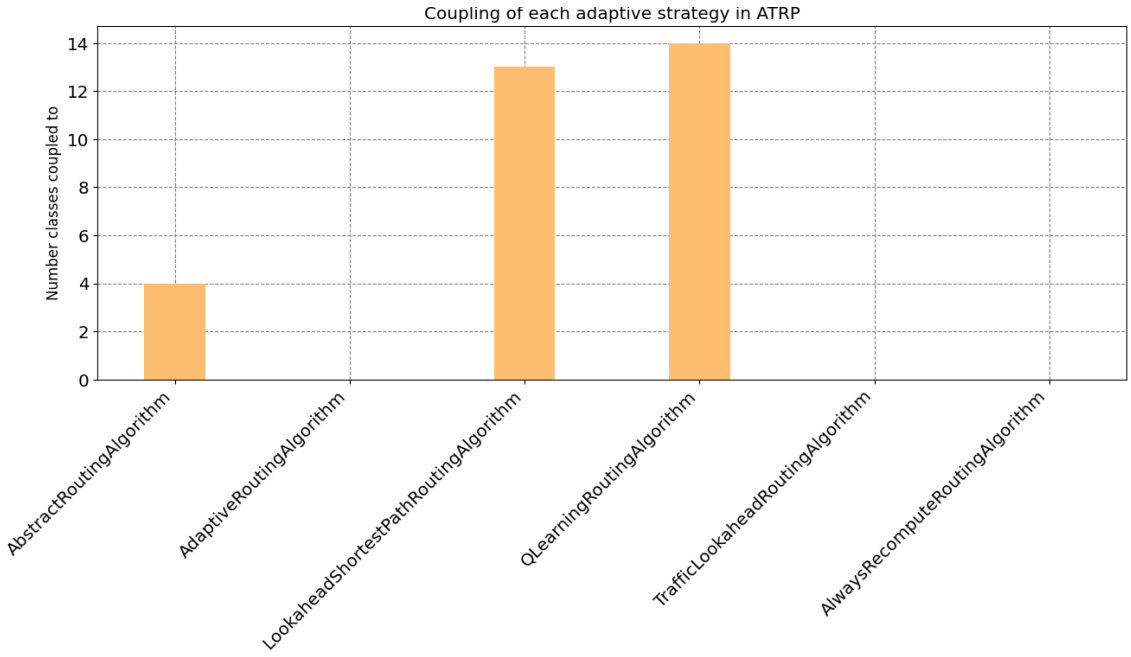

Figure 17: Graph showing the Coupling of each ATRP adaptive strategy, with the y-axis representing the number of components in the system it is coupled to and the x-axis representing the names of the adaptive strategies.

As can be seen from Figure 17, the `LookaheadShortestPathRoutingAlgortihm` and `QLearningRoutingAlgorithm` are the adaptive strategies in ATRP that have the highest coupling. Likewise for the TRAPP exemplar, Figure 18 shows that the `TunePlanningResolution` strategy has the highest coupling.

For these strategies, it indicates that they are externally more connected than others. When it comes to the modifiability of a SAS, it would be optimal for its adaptive strategies to score low in this metric, since it means that making a change in one of them would require less changes in other components in the system [12].

In contrast to this, strategies such as `TrafficLookaheadRoutingAlgorithm`, `AdaptiveRoutingAlgorithm`, and `AlwaysRecomputeRoutingAlgorithm` are not coupled to any other classes at all, making them a more accessible target for modification.

Additionally, Figure 19 presents the Lack of Cohesion of each adaptive strategy in ATRP, where the x-axis represents the name of the adaptive strategy and the y-axis represents the average number of class instance variables that are not used by its class' methods. Understand does not provide a means to measure the Lack of Cohesion in Python implementations, which is why only the results from ATRP could be collected.

In regards to the results of Figure 19, the same adaptive strategies that have high coupling in ATRP also have a high Lack of Cohesion. Similar to Coupling, it is considered better to score low in this metric, since it implies that a given class is internally well-

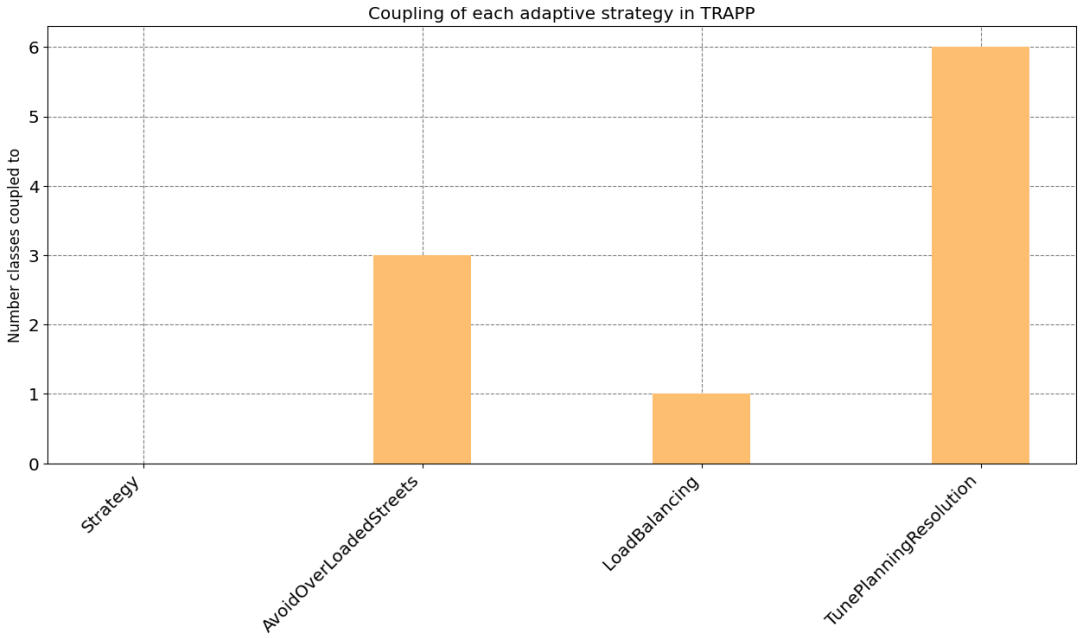

Figure 18: Graph showing the Coupling of each TRAPP adaptive strategy, with the y-axis representing the number of components in the system it is coupled to and the x-axis representing the names of the adaptive strategies.

connected and therefore more cohesive [12].

Alternatively, the `AdaptiveRoutingAlgorithm`, `TrafficLookaheadRoutingAlgorithm` and `AlwaysRecomputeRoutingAlgorithm` each have zero Lack of Cohesion. This is due to the fact that neither of these strategies have any class instance variables.

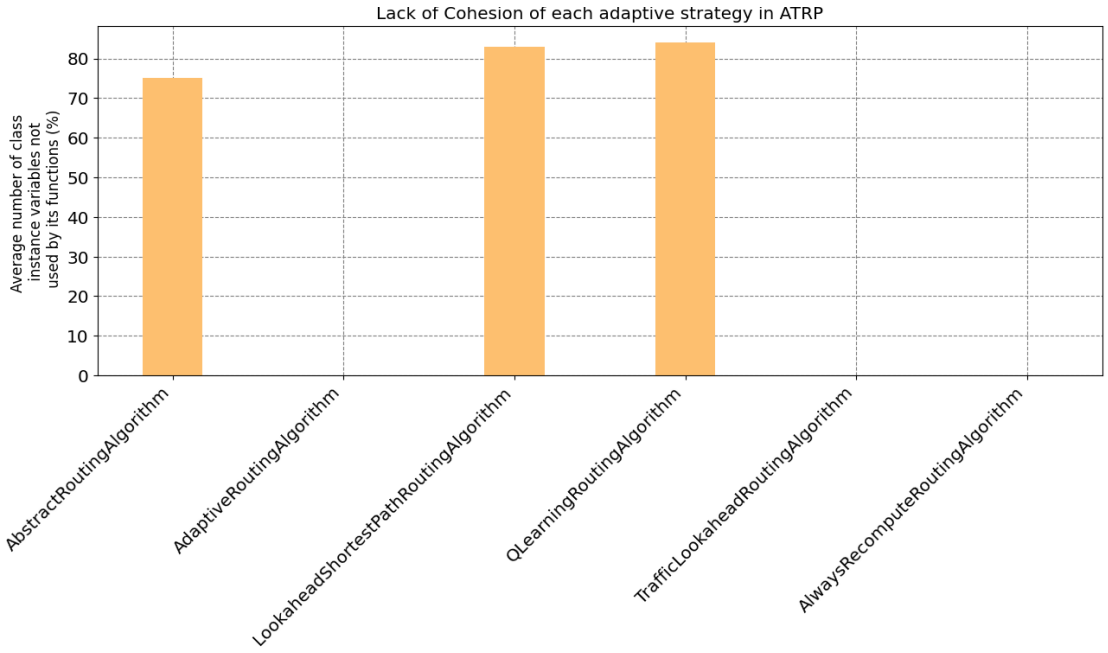

Figure 19: Graph showing the Lack of Cohesion of each ATRP adaptive strategy, with the y-axis representing the average number of class instance variables that are not used by its methods and the x-axis representing the names of the adaptive strategies.

Limitations and Future Work. With respect to modifiability, there are missing elements that may be considered important that can provide more information on how changeable the adaptive strategies are in a SAS. One such part, as can be seen in the SIG Maintainability Model, is code duplication, which may add to unnecessary complexity in regards to maintainability.

Therefore, future work could include examining how much duplicated code exists in the codebase of a SAS. Understand does not provide a means to compute this, but it is a task that is fully automatable [12]. A more useful solution could also take this a step further by identifying a degree of duplication, which would involve studying how similar pieces of code are to each other [25]. If the similarity between two code fragments is highly apparent, then there may exist a way to refactor them to reduce the degree of duplication, thereby improving maintainability.

6.5 Summary

So far, this section has covered the details related to the evaluation of two case study implementations using ASMS. It has presented several technical details related to the source code properties of each exemplar in an effort to provide further context to the evaluation results. It has also covered how the adaptive strategies have been identified, as well as the way in which ASMS has been applied to them in order to collect the results.

Additionally, the second half of this section has introduced the setup behind the evaluation and showed why the adaptive strategies of the ATRP and TRAPP exemplar have been evaluated on. From there, it has presented the results of the metrics in the form of graphs and examined its more compelling findings. On top of that, it has also discussed several limitations for each metric based on their results and considered possible future work that may resolve some of these identified restrictions.