Quantifying the Relationship Between Software Design Principles and Performance in Jason: a Case Study with Simulated Mobile Robots

Patrick Gavigan and Babak Esfandiari

Carleton University, Ottawa, Canada
{patrickgavigan,babak}@sce.carleton.ca

Abstract. We investigated the relationship between various design approaches of AgentSpeak code for Jason Beliefs-Desires-Intentions (BDI) agents and their performance in a simulated automotive collision avoidance scenario. Also explored was how the approaches affected software maintainability, assessed through coupling, cohesion, and cyclomatic complexity. We then compared each agent's performance, specifically their reasoning cycle duration and their responsiveness. Our findings revealed that agents with looser coupling and higher cohesion are more responsive to stimuli, implying that more maintainable AgentSpeak can result in better performing agents. Performance was inversely related to cyclomatic complexity.

1 Introduction

While there has been a great deal of work in the Agent-Oriented Software Engineering (AOSE) community on the design of Multi-Agent Systems (MAS), there doesn't seem to be much guidance in terms of how software design principles should be applied to writing individual agents using AgentSpeak. Different approaches have trade-offs in terms of their software maintainability and their performance. This work aims to study this trade-off in the context of a simulated autonomous car developed with Jason [2][11]. As in [14][20], the agents drove the car on a street toward an obstacle and needed to stop in time to avoid a collision. There is a high number of perceptions being generated from the car's sensors, which is a factor in the performance of the agent.

One possible design approach, not necessarily recommended, would be to write AgentSpeak code exclusively with goal-directed behaviours - plans that are only triggered by achievement goals. Another extreme design approach would be to write exclusively reactive behaviour using only belief-triggered plans. A more *idiomatic* approach would be to balance the design of the agent's behaviour using a combination of goal-triggered plans and belief-triggered plans to reduce the scope of concerns that each plan needs to handle, hopefully improving maintainability. A judicious use of custom event and option selection functions can also help reduce the number of guards in plans and the number of candidate plans, as well as reduce the dependency between plans and their ordering.

To assess the maintainability of these approaches, we adapt software engineering metrics from the object-oriented design field aiming to evaluate coupling, cohesion, and complexity. These concepts and the specific metrics are detailed later in this paper.

To assess the performance impact of the various design approaches, and the possible trade-offs with maintainability criteria, we measure the duration of the agent's reasoning cycle, the time it took for the agent to decide to stop to avoid the collision, and the number of collisions that occurred for each agent.

In the remainder of this paper we first look at related work that has explored software engineering metrics in the AOSE field as well as work on the performance of Beliefs-Desires-Intentions (BDI) agents. This is followed by a discussion of the experiment setup and the alternative agent designs. These alternative designs are then assessed using the adapted concepts of coupling, cohesion and complexity. Next, these alternative designs are tested in the collision avoidance scenario. Finally, the noted performance difference is investigated with a profiler to identify the underlying cause.

2 Background and Related Work

Although there are a number of metrics used by the field of AOSE, these metrics tend to focus on the interactions between multiple agents and less about the design of individual agents. Examples of these include social ability, autonomy, and proactivity [23]. Another theme in AOSE is the development of design methodologies. Wooldridge et. al. provided Gaia [27] which focuses on the roles of agents within a MAS in terms of concepts such as permissions, responsibilities, protocols, activities, liveness properties, and safety properties. Kinny et. al. [12] propose a process that examines roles and responsibilities of agents, plans that may be used to achieve them, and belief structure of the system. Miles et. al. [17] noted that object-oriented software provides highly cohesive and separated components that can be modified individually, and highlighted that this type of design would benefit the design of agent software.

The object-oriented community assesses maintainability using the concepts of coupling and cohesion [25][1]. Coupling refers to how interconnected a class is with another class. When software has loose coupling, changes to one class don't break other classes and force additional changes. The connections between classes should follow well-defined interfaces and not be over-reliant on the implementation-specific details of these interfaces. Cohesion refers to the number of tasks or responsibilities of each class or method. For software to have high cohesion, the modules, classes, or methods should each be responsible for one, well defined, task or concern. Having higher cohesion increases the likelihood that the code can be reused. Although there is some precedent for using these metrics with MAS, specifically in terms of the connections between agents, there does not seem to be widespread adoption of these types of metrics for assessing the connections and responsibilities of the plans within an agent [10]. There is also precedent for using these metrics to assess declarative software implemented in languages such as Prolog [13] [21], which has a syntax similar to AgentSpeak. Another useful metric is McCabe's [15] *Cyclomatic complexity* which is calculated using the number of edges and nodes in a program's control graph and the number of connections it has to external modules. Naturally it is preferable for software to be less complex. This has also been used in the context of MAS, again focusing on the connections between agents [4] and has been used for evaluating declarative software in Prolog [19].

There has been effort to examine the performance of BDI agents. Stabile and Sichman [24] found that depending on the number of perceptions Jason handles, belief update and unification accounted for up to 99% of execution time. Miller and Esfandiari [18] investigated the time complexity of Jason's reasoning cycle components. Through profile testing they confirmed their analysis and found that increasing the number of perceptions resulted in a polynomial growth in execution time and that increasing the number of beliefs and plans resulted in a linear growth in execution time. Concerned about Stabile and Sichman's findings, Pantoja et. al. tested their ARGO agent's performance in terms of its responsiveness to stimuli by having it drive a car toward a wall and observed the stopping performance [20][14]. They found that perception filtering removed the need for the agent to update beliefs from irrelevant sensors and led to improved performance. Cardoso et. al. [3] used the same approach to prevent their agent from being overwhelmed.

Wesz [26] compared agent behaviour code implemented using Jason with JaCaMo to similar behaviours implemented in Python and found that the Jason agents generally had fewer lines of code, interpreting this to mean that the Jason code was less complex. Although the number of lines of code can provide some insight, the metrics of coupling and cohesion [25][1] are preferred for assessing maintainability and the cyclomatic complexity [15] metric is preferred for assessing complexity.

3 Methodology

Our study of the trade-offs between the design of agents using AgentSpeak and their performance requires both a test environment in which their performance can be assessed and a set of alternative designs to compare. This section provides the details of the collision avoidance experiment, including the performance metrics measured. This is followed by the design details for each of the alternative agents.

3.1 Collision Avoidance Experiment Setup

In our test, each agent drove the car toward an obstacle to trigger its obstacle avoidance behaviour to stop the car. There are several ways to observe the differences in agent performance. For example, the agents' reasoning times and reaction to stimuli can be measured, as in [20][14]. The experiment was as follows:

- Measure the reasoning rate and perception rate. Was the agent keeping up?

- Measure the time from the observation of the obstacle to the stop action.
- Observe if the collision was avoided or not.
- Compare the results of different agent designs. Was there a difference?

AirSim's automotive neighbourhood was used for our experiment as it provided a safe and repeatable test environment. AirSim is an open-source high fidelity simulator maintained by Microsoft [16][22]. To perform the experiments, Jason was connected to AirSim using SAVI ROS BDI [9], meaning that the only difference between the agents was their internal implementation. The agent had access to a variety of sensors and actuators, including a Global Positioning System (GPS) sensor, a compass, a speedometer, an obstacle detection laser imaging, detection, and ranging (LIDAR), a lane-keep assist camera, and a cruise controller for controlling the throttle and breaks. Sensor data from each of the sensors is provided to the agent for every reasoning cycle. Our software is available on GitHub [6] and videos of the car driving are available on YouTube [5][7][8]. These videos include examples of the car stopping before the obstacle [8], crashing into the obstacle [7], and an additional video of the car swerving to avoid the obstacle and continue driving to the end of the street to its destination [5].

3.2 Agent Designs

A set of design alternatives are needed for measuring the trade-offs between design principles and performance. As mentioned earlier, two extreme design approaches would be to write AgentSpeak plans using only goal triggers and plans using only belief triggers, yielding exclusively goal-directed or reactive agents. Although these design approaches are somewhat artificial and not necessarily how agents may be designed in practice, it was noticed anecdotally that novice AgentSpeak programmers would take these approaches. By using these extremes it also enables differences in performance and software properties to be observed. Designing more idiomatically would include balancing the use of both goal-triggered and belief-triggered plans. This approach was used in concert with a behaviour prioritization scheme which prioritizes safety-related event triggers over others and selects non-default plans over default plans. The last alternative, which provides context for the results for the BDI agents in comparison to more traditional software design was an imperative program written in Python. The following paragraphs and listings detail the designs of these alternatives and the prioritization scheme.

Behaviour Prioritization The idiomatic agent makes use of a behaviour prioritization scheme which selects events related to safety over events related to movement. It also prioritizes non-default plans over default plans. These two features are expected to reduce the number of plan guards, the number of candidate plans associated with each event trigger, and the dependency on the relative ordering between plans. This is further expected to reduce the scope of concerns that each plan needs to handle, hopefully improving maintainability.

The selection of safety related plans over movement plans was implemented by overriding the event selection function in Jason's Agent class. The customized methods are provided in listing 1. In order for the function to select the highest priority event, it needs to have knowledge of the types of behaviour that each event triggers. This was provided by a set of prioritization beliefs, which specified if the events were tied to safety or movement behaviour. To select the highest priority event, the `getHighestPriorityEvent` method queries the belief base for these prioritization beliefs to identify the priority level of each trigger in the queue. These are compared to a list of event priorities, which provides the relative ranking.

Listing 1: Event Selection.

```
protected List<String> eventPriorities = Arrays.asList("safety", "movement");
public Event selectEvent(Queue<Event> events) {
    Event selected;
    if (events.size() > 1) {
        selected = this.getHighestPriorityEvent(events);
    } else {
        selected = super.selectEvent(events);
    }
    events.remove(selected);
    return selected;
}
protected Event getHighestPriorityEvent(Queue<Event> events) {
    BeliefBase beliefs = this.getBB();
    for (String priority : eventPriorities) {
        for (Event event : events) {
            String trigger = event.getTrigger().getLiteral().getFunctor();
            for (Literal belief : beliefs) {
                if (belief.getFunctor().contentEquals(priority)) {
                    List<Term> terms = belief.getTerms();
                    for (Term term : terms) {
                        if (term.toString().contentEquals(trigger)) {
                            return event;
                        }
                    }
                }
            }
        }
    }
    return super.selectEvent(events);
}
```

The prioritization of non-default plans over default plans was implemented by overriding the option selection function in Jason's Agent class. Shown in listing 2, the method checks the length of the context for all of the available options. It returns the first available option that does not have an empty context. If no alternatives are available the default plan is returned.

Listing 2: Option Selection.

```
@Override
public Option selectOption(List<Option> options) {
    Option selected;
    if (options.size() > 1) {
        selected = this.getHighestPriorityOption(options);
    } else {
        selected = super.selectOption(options);
    }
    options.remove(selected);
    return selected;
}
protected Option getHighestPriorityOption(List<Option> options) {
    Option priorityOption = null;
    Iterator<Option> optionInstance = options.iterator();
    boolean first = true;
    while (optionInstance.hasNext()) {
        Option current = optionInstance.next();
        Plan currentPlan =  current.getPlan();
        LogicalFormula context = currentPlan.getContext();
        if (first) {
            priorityOption = current;
            first = false;
        }
        if (context != null) {
            priorityOption = current;
            break;
        }
    }
    return priorityOption;
}
```

Supporting Rules To drive the car, the different BDI agents were supported by the rules in listing 3. The first six rules set the threshold for the collision avoidance behaviour, calculate if the car is near or at a particular location (for navigation purposes), calculate range and bearing, and calculate course corrections using the compass. The next three rules calculate the steering setting based on the degree of course correction needed. These rules set the steering setting to the maximum magnitude when the course correction was greater than 20°. If the magnitude was smaller, the steering setting was dampened by dividing it by 180, a crude but effective way of controlling steering. The last rule extracts the steering setting from a lane-keep perception if the lane-keep assist module was successful at detecting the lane.

Listing 3: Localization and Steering Rules.

```
obstacleStop :- obstacle(Distance) & Distance < 7.0.
nearLocation(Location, Range) :- gps(CurLat,CurLon) & locationName(Location,
    [Lat,Lon]) & navigation.range(CurLat,CurLon,Lat,Lon,Range) & Range < 20.
atLocation(Location, Range) :- gps(CurLat,CurLon) & locationName(Location,
    [Lat,Lon]) & navigation.range(CurLat,CurLon,Lat,Lon,Range) & Range < 7.
destinationRange(Location,Range)
    :- locationName(Location,[DestLat,DestLon]) & gps(CurLat,CurLon)
    & navigation.range(CurLat,CurLon,DestLat,DestLon,Range).
destinationBearing(Location,Bearing)
    :- locationName(Location,[DestLat,DestLon]) & gps(CurLat,CurLon)
    & navigation.bearing(CurLat,CurLon,DestLat,DestLon,Bearing).
courseCorrection(TargetBearing, Correction)
    :- compass(CurrentBearing) & declanation(Declanation)
    & (Correction = TargetBearing - (CurrentBearing + Declanation)).
steeringSetting(TargetBearing,1)
    :- courseCorrection(TargetBearing,Correction) & (Correction >= 20).
steeringSetting(TargetBearing,-1)
    :- courseCorrection(TargetBearing,Correction) & (Correction <= -20).
steeringSetting(TargetBearing,Correction/180) :- courseCorrection(
    TargetBearing,Correction) & (Correction < 20) & (Correction > -20).
lkaSteering(Steering)
    :- lane(Steering,A,B,C,D) & ((not (C == 0)) | (not (D == 0))).
```

Idiomatic Agent The idiomatic agent used separate triggers for collision avoidance, maneuvering, steering, and speed control. It also used the behaviour prioritization scheme discussed earlier, which prioritized collision avoidance over other plans. Without this the plans would need to be provided in order of their relative priority or include mutually exclusive contexts which, although complicates maintainability, seems to be fairly common practice. Listing 4 provides the plans, triggered by the `!waypoint(_)` goal, responsible for driving the car using lane and compass following. This goal was identified for the event selection function using the `movement(waypoint)` belief. Except for the first plan, for the case where the car had arrived, the plans each readopted `!waypoint(_)` to continue driving the car to the destination. The second slowed the car as it approached its destination and the third was applicable when the car was not near its destination. In this case the lane-keep assist was enabled and the speed was set to its cruising speed. The last of the `!waypoint(_)` plans is a default plan which maintained recursion if the other plans weren't applicable. Collision avoidance was implemented separately using a high priority atomic belief-triggered plan. This plan triggered when the LIDAR sensor detected an obstacle and stopped the car when the distance to the obstacle was less than its threshold. The customized event selection function identified this as a safety related event using the `safety(obstacle)` belief, enabling it to select this event over others, If this plan was applicable, it would be set as the agent's intention.

Listing 4: Idiomatic Agent Driving Plans.

```
movement(waypoint).
+!waypoint(Location) : atLocation(Location,_)
    <- !controlSpeed(0); !controlSteering(0,lkaOff).
+!waypoint(Location) : nearLocation(Location,_) & (not atLocation(Location,_))
    & destinationBearing(Location,Bearing) <- !controlSteering(Bearing,
    lkaOff); !controlSpeed(3); !waypoint(Location).
+!waypoint(Location)
    : (not nearLocation(Location,_)) & destinationBearing(Location,Bearing)
    <- !controlSteering(Bearing,lkaOn); !controlSpeed(8); !waypoint(Location).
+!waypoint(Location) <- !waypoint(Location).
safety(obstacle).
@obstacleAvoidance [atomic]
+obstacle(Distance) : obstacleStop <- !controlSpeed(0).
```

Speed and steering settings were handled by the plans in listing 5. The `+!controlSpeed(_)` plans used a belief so that the speed wasn't reset needlessly. The first plan updated the speed if a change was needed, followed by the second which handled the case where the belief was present. The cruise control, which controlled the accelerator and brake, was commanded using `setSpeed(_)`. The last `+!controlSpeed(_)` plan was the default, applicable when no change was needed. Also provided are the steering plans. The first of these was for steering the car with magnetic bearing angles, useful if the lane-keep assist was disabled or the lane was not detected. The second plan used the lane-keep assist to follow the road. Lastly the default plan is provided.

Listing 5: Speed and Steering Control.

```
+!controlSpeed(Speed) : speedSetting(Old) & (Old \== Speed)
    <- -speedSetting(_); +speedSetting(Speed); setSpeed(Speed).
```

```
+!controlSpeed(Speed) : not speedSetting(_)
    <- +speedSetting(Speed); setSpeed(Speed).
+!controlSpeed(_).
+!controlSteering(Bearing,LkaSetting) : steeringSetting(Bearing,Steering)
    & ((not lkaSteering(_)) | (LkaSetting == lkaOff)) <- steering(Steering).
+!controlSteering(Bearing,lkaOn) : lkaSteering(Steering)
    <- steering(Steering).
+!controlSteering(_,_).
```

Goal-Directed Agent Listing 6 provides the goal-directed agent, which only used plans triggered by achievement goals and didn't use the prioritization scheme. Without the prioritization the plans needed to be provided in order of their relative priority and have mutually exclusive contexts, otherwise the reasoner would select the first applicable plan it found rather than the most appropriate. The driving plans were triggered by `!waypoint(_)`. The first plan stopped the car at its destination, followed by a plan that stopped the car for an obstacle. Next were two plans responsible for steering the car with either the lane-keep, if available, or the compass. The agent controlled speed and steering using the plans in listing 5.

Listing 6: Goal-Directed Agent Decision Making.

```
+!waypoint(Location) : atLocation(Location,_)
    <- !controlSpeed(0); !controlSteering(0,lkaOff).
+!waypoint(Location) : obstacleStop <- !controlSpeed(0).
+!waypoint(Location) : nearLocation(Location,_) & (not atLocation(Location,_))
    & destinationBearing(Location,Bearing) & (not obstacleStop) <-
    !controlSteering(Bearing,lkaOff); !controlSpeed(3); !waypoint(Location).
+!waypoint(Location) : (not nearLocation(Location,_))
    & destinationBearing(Location,Bearing) & (not obstacleStop)
    <- !controlSteering(Bearing,lkaOn); !controlSpeed(8); !waypoint(Location).
+!waypoint(Location) <- !waypoint(Location).
```

Reactive Agent The reactive agent's plans, in listing 7, were triggered by the LIDAR beliefs at each reasoning cycle. Similar to the goal-directed agent these plans were listed in descending order of priority and had mutually exclusive contexts to ensure the correct one was selected. The first two plans stopped the car for collision avoidance and when the car was at its destination followed by the third plan for slowing the car near the destination. The last two plans handled steering using the lane-keep assist or the compass. The reactive agent did not use the speed or steering control sub-goals as this agent was implemented without goals. Instead, it used the `setSpeed(_)` action directly.

Listing 7: Reactive Agent Decision Making.

```
+obstacle(Distance) : obstacleStop <- setSpeed(0).
+obstacle(Distance) : (not obstacleStop) & navigate(Location)
    & atLocation(Location,_) <- setSpeed(0).
+obstacle(Distance) : navigate(Location) & nearLocation(Location,_)
    & (not atLocation(Location,_)) & destinationBearing(Location,Bearing)
    & (not obstacleStop) & steeringSetting(Bearing, Steering)
    <-  steering(Steering); setSpeed(3).
+obstacle(Distance) : navigate(Location) & (not nearLocation(Location,_))
    & destinationBearing(Location,Bearing) & (not obstacleStop)
    & lkaSteering(Steering) <- steering(Steering); setSpeed(8).
+obstacle(Distance) : navigate(Location) & (not nearLocation(Location,_))
    & destinationBearing(Location,Bearing) & (not obstacleStop)
    & (not lkaSteering(_)) & steeringSetting(Bearing, Steering)
    <- steering(Steering); setSpeed(8).
```

Imperative Agent The last alternative, in listing 8, used imperative software in Python instead of the BDI reasoner. When data was received, the script would drive by following the lane and stopping when an obstacle was observed within range. The first check assessed if the car was near an obstacle and stopped if needed. The next determined if the car had started driving and set the the controller's speed. The last option handled steering using the lane-keep assist or compass steering. The function for calculating the compass steering setting is also provided in the listing.

Listing 8: Imperative Agent Decision Making.

```
def decide(gps, compass, lane, speed, obstacle):
    global stopRange, currentSpeedSetting, speedSetPoint
    action = ’’
    if (obstacle < stopRange):
        action = ’setSpeed(0)’
    elif currentSpeedSetting == 0:
        currentSpeedSetting = speedSetPoint
        action = ’setSpeed(’ + str(currentSpeedSetting) + ’)’
    else:
        (lkaSteering,_,_,c,d) = lane
        if ((c != 0) or (d != 0)):
            action = ’steering(’ + str(lkaSteering) + ’)’
        else:
            compassSteering = getCompassSteering(gps,compass)
            action = ’steering(’ + str(compassSteering) + ’)’
    if action != ’’:
        act(action,actionsPublisher)
        sendMessage(action,outboxPublisher)
def getCompassSteering(gps,compass):
    global destination, wgs84, declanation
    (curLat,curLon) = gps
    current = wgs84.GeoPoint(latitude=curLat, longitude=curLon,
        degrees = True)
    destinationBearing = destination.delta_to(current).azimuth_deg[0]
    courseCorrection = destinationBearing - (compass + declanation)
    if courseCorrection >= 20:
        steeringSetting = 1
    elif courseCorrection <= -20:
        steeringSetting = -1
    else:
        steeringSetting = courseCorrection/180
    return steeringSetting
```

4 Assessment of Software Engineering Properties

Discussed in section 2, maintainable object-oriented software tends to have *loose coupling* and *high cohesion* [1][25]. Shifting focus to MAS, although these metrics have been applied to the interactions between agents, there does not seem to be an accepted definition for examining the connections between plans within an agent. Therefore, the spirit of these metrics needed to be adapted. This adaptation, and the assessment of coupling and cohesion for the agents studied in this paper, are discussed in section 4.1.

The assessment of software complexity uses McCabe's *cyclomatic complexity*, based on the number of possible paths through a program [15]. As was the case with coupling and cohesion, there was limited use of this metric for assessing AgentSpeak code. The definition of this metric and its use for assessing the agents studied in this paper is provided in section 4.2.

4.1 Assessment of Coupling and Cohesion

Consider what would make for loose coupling between plans; changes made to plans with loose coupling should not require changes to other plans, especially not plans triggered by different events. To isolate changes to plans from each other, each triggering event should trigger the fewest possible number of plans, ideally with fewer terms in their contexts. Designing this way reduces the likelihood of changes to one plan affecting many others, as there are fewer plans and terms that could require changing. Plans should be implemented without concern for the order in which they are provided in the plan base, nor what other goals the agent may have. Otherwise, changes to the order of the plans could result in changes in the agent's behaviour.

Shifting to cohesion, plans that have high cohesion should only be responsible for a single concern. For example, a triggering event for plans that implement steering and speed control, rather than only one of those two concerns, would have lower cohesion. One possible way to increase cohesion could be to use subgoals. With each event triggering fewer behaviours there should not need to be many plans per trigger. Beliefs should only be maintained by plans triggered by the same triggering event, separating the concern for these beliefs to specific plan sets. This separation of concerns between the different behaviours encapsulates them within the plans that are triggered by each event.

Based on the discussion above, several properties were used to assess the coupling and cohesion of alternative agents in this study. These are enumerated below. The measurements of these properties for that alternative designs are provided in table 1.

1. The number of triggering events in the program,
2. The number of plans associated with each triggering event,
3. The number of logic expressions joined by conjunction in the plan contexts (which was the format used by all the plans in this study), and
4. The number of concerns addressed by the plans triggered.

The idiomatic agent has four different event triggers, each responsible for a single concern – either avoiding an obstacle, controlling the car, controlling the speed, or controlling the steering. The plans that implement these concerns have relatively simple contexts, consisting of conjunctions of fewer logic expressions than the alternatives. It also has fewer plans per trigger.

Table 1: Properties of BDI Agent Alternatives.

		Idiomatic	Goal-Directed	Reactive
# Triggering Events		4	3	1
# Plans per Event	Obstacle	1	–	5
	Waypoint	4	5	–
	ControlSpeed	3	3	–
	ControlSteering	3	3	–
# Logic Expressions per Context	Obstacle	1/1=1	–	21/5 = 4.2
	Waypoint	6/4=1.5	9/5 = 1.8	–
	ControlSpeed	3/3=1	3/3=1	–
	ControlSteering	3/3=1	3/3=1	–
# Concerns per Trigger	Obstacle	1	–	4
	Waypoint	1	2	–
	ControlSpeed	1	1	–
	ControlSteering	1	1	–

The goal-directed agent only used achievement goal-triggered plans, meaning that the collision avoidance needed to be handled by the waypoint plans. As a result, the waypoint plans had two concerns to manage rather than one, increasing the number of logic expressions joined by conjunctions in the plan contexts and the number of triggered plans. There was another important difference between the goal-directed and idiomatic agents – the goal-directed agent did not have access to the behaviour prioritization scheme, meaning that the plans needed to be listed in order based on their relative priority with each other or have mutual exclusion guaranteed in their contexts. This dramatically increased the coupling between the plans.

The reactive agent also did not use the prioritization scheme, meaning that it needed to have mutual exclusion between the plan contexts and the plans needed to be listed in their relative priority. This version defined plans exclusively using belief-triggers which resulted in the most complicated of the BDI agents, where a single event trigger was associated with all of the aspects of controlling the car. The five plans that were implemented had relatively complicated contexts, far more so than either of the alternative agents.

In addition to the BDI agents, an imperative version of the car controller was written using a Python function which was made up of a set of nested conditional statements. This function provided an action for the car given a set of perceptions, and was responsible for controlling the speed, steering, and avoiding collisions. The conditional statements in this function had a depth of three, making this a rather complicated function with relatively poor cohesion. However, as it did not interact with other modules, the coupling metric was not applicable.

Table 2 provides a relative comparison of the different implementations in terms of their performance on coupling and cohesion based on the previously discussed plan attributes. The idiomatic agent scored highest on both coupling and cohesion. This was for several reasons: the plans could be provided in any order, each triggering event was tied to a single responsibility, and the resulting plans had fewer plans per trigger and fewer logic statements in their plan contexts. The goal-directed agent was second place for both coupling and cohesion, ranked behind the idiomatic agent. For coupling this was because of the lack of prioritization of behaviour, meaning that the plans needed to be listed in relative priority while ensuring mutual exclusion. This resulted in more logic statements in the plan contexts and an additional plan for `!waypoint`. For cohesion this was because the concerns for collision avoidance were combined with the concern for maneuvering the car. The imperative version did not have any connections with other modules and therefore the coupling metric was not applicable. The imperative version did however score poorly on cohesion, as it was implemented with a nested conditional statement that needed to address all the behaviours of the agent. If any changes were required, the nested conditional statements would need to be unravelled and redesigned. The worst scores were held by the reactive agent, which had very poor coupling and cohesion – any change in one plan would likely require changes to many other plans. These plans were the most complex with the most logic expressions in the plan contexts, most concerns per trigger, and most plans per event. This was scored worse than the imperative agent as these behaviours were implemented in separate plan blocks: at least in the imperative agent the conditional statement was contained in a single block.

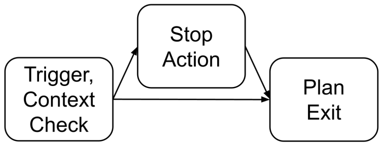
Table 2: Relative Rank for Coupling and Cohesion (Best: 1, Worst: 4).

Version	Idiomatic	Goal-Directed	Reactive	Imperative
Coupling	1	2	3	N/A
Cohesion	1	2	4	3

4.2 Assessment of Cyclomatic Complexity

McCabe's [15] *cyclomatic complexity* is based on the number of possible paths through a program. It is defined by $M = E - N + 2P$, where the parameters are properties of a program's control graph where E is the number of edges, N is the number of nodes, and P is the number of connected components. The goal should be to have software that has low cyclomatic complexity.

To assess the cyclomatic complexity of the different agents it was first necessary to draw the node graphs for each of the components of the agents. The nodes in the control graph represent lines of AgentSpeak code and the edges are execution paths between those lines through the different triggered plans. An example of these node graphs, and the resulting analysis, is provided for the idiomatic agent's collision avoidance plan in figure 1. In this case, the plan is triggered on the perception of `obstacle(_)`. There are two execution paths for this component: either the obstacle is in range and the plan is applicable and the stop action is run, or it is not and the plan is not executed. The connected component was the behaviour prioritization scheme.



Property	Value
# Edges (E)	3
# Nodes (N)	3
# Connected Components (P)	1
Cyclomatic Complexity (M)	2

Fig. 1: Node Graph of Idiomatic Agent's Collision Avoidance Behaviour.

Table 3 summarises this analysis for the alternative designs. The reactive and imperative agents have the lowest cyclomatic complexity because the goal-directed and idiomatic agents were made up of more components, such as their sub-goals for controlling speed and steering. The idiomatic agent had higher cyclomatic complexity than the goal-directed agent because of its connection to the prioritization scheme, however, having this additional functionality allowed the plans for the idiomatic agent to be in any order without concern for mutual exclusion. The goal-directed and reactive agents needed to have their plans provided in order from highest to lowest priority for the default event and option selection functions to select the most appropriate plan to run. Furthermore, the goal-directed, reactive, and imperative agents all had more nodes and edges in their graphs, making the maintenance of those graphs likely more complicated than the maintenance of the idiomatic agent's graphs.

4.3 Software Engineering Properties Summary

The agent designed with only reactive plans had the lowest overall cyclomatic complexity, but scored poorly on cohesion and coupling. The imperative agent had similar cyclomatic complexity to the reactive agent but again it scored poorly on coupling and cohesion. The idiomatic agent actually had the worst cyclomatic complexity scores, primarily due to the connections it had to the prioritization scheme, however it scored very well on coupling and cohesion compared to the alternatives. This was because the idiomatic agent was able to keep each triggering topic focused on a single concern, with fewer conditional terms needed in the context guards for those plans. This indicates that the idiomatic agent used good design practices. The idiomatic agent also had the advantage of allowing the plans to be implemented in any order, unlike the alternatives, which either had to ensure that the plans had mutually exclusive contexts or had to ensure that they were provided in order of their relative priority.

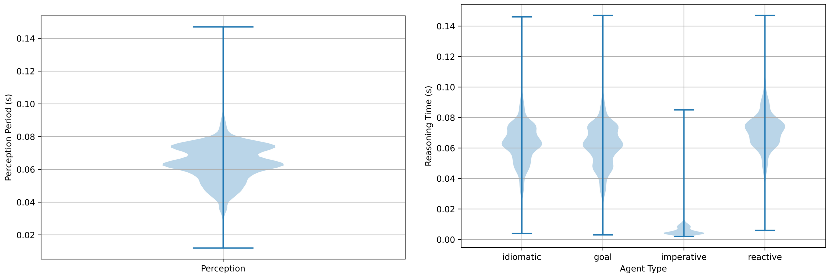
Table 3: Summary of Cyclomatic Complexity Parameters.

Graph	# Edges (E)	# Nodes (N)	Names of Connected Components	# Connected Components (P)	Cyclomatic Complexity ($M = E - N + 2P$)
Idiomatic Collision Avoidance	3	3	Prioritization	1	2
Idiomatic Waypoint	13	11	Steering Goal Speed Goal Prioritization	3	8
Idiomatic Speed	8	7	Prioritization	1	3
Idiomatic Steering	5	4	Prioritization Steering Rules	2	5
Goal-Directed Speed	8	7	–	0	1
Goal-Directed Steering	5	4	Steering Rules	1	3
Goal-Directed Waypoint	15	12	Steering Goal Speed Goal	2	7
Reactive	14	10	–	0	4
Imperative	22	17	–	0	5

5 Runtime Performance

Each of the alternative agents were tested on a desktop computer running Windows 10 with an Intel Core i7-5820K CPU @ 3.30 GHz with 64 GB of system RAM and an NVIDIA GTX 970 with 4 GB of RAM. To assess their relative performance, the reasoning time for each of these agents was measured and are reported in figure 2. For the imperative version, which does not have a reasoner, the reasoning time refers to the execution time for each of its decision cycles. It is clear that the imperative version was able to make decisions much faster than the BDI agents. The next fastest reasoning times are for both the idiomatic and goal-directed agents, which appear to have very similar reasoning performance. These reasoning times also appear to be inline with the perception period. The slowest was the reactive agent, which appears to have had difficulty keeping up with the perceptions. The reason for this difficulty was investigated and is discussed later in this section.

Although the results provided in figure 2 provide some insight into the reasoning performance of the agents, these results are still somewhat abstract. Ultimately, it is more intuitive to examine how these differences in reasoning affect the agent's performance in an environment. To do that, the agents were observed



(a) Perception period.

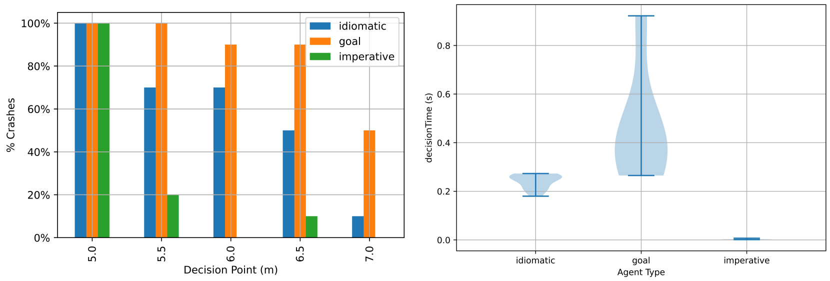
(b) Reasoning time.

Fig. 2: Agent Perception Period and Reasoning Times.

in a collision avoidance scenario where the agent drove a simulated car toward an obstacle to trigger an avoidance maneuver. The decision threshold for this maneuver was varied and tested for the different agents. The effect of this was that the agent would have a different amount of time to react to the obstacle and stop. The two properties of the runs that were observed included if the car crashed and the time needed for the agent to decide to stop once it was close enough to the obstacle. Each agent was run through the scenario 10 times for each decision threshold.

Figure 3 shows a clear difference in performance between the different agents. None of the agents were able to prevent a crash with a decision point of 5m. As the decision point was increased in increments of 0.5 m, differences in performance started to become evident. The imperative agent, which was previously observed to have the fastest decision time, avoided crashes in almost every subsequent test. This was significantly better than the BDI agents. Between the BDI agents, there were also clear differences in performance. Keen readers will notice that the reactive agent is not shown in the graphs. This is because the reactive agent was not able to keep up with the sensory perceptions. The result was that it was not able to stay on the road and crashed into a fence next to the road before reaching the decision point. This occurred in every test, leaving the idiomatic and the goal-directed agent. Between these two, the idiomatic agent was successful at avoiding more crashes than the goal-directed agent. This was also evident when looking at the decision time for the agents, where the goal-directed agent took the longest to make the decision to stop and the imperative agent was by far the fastest to decide to stop.

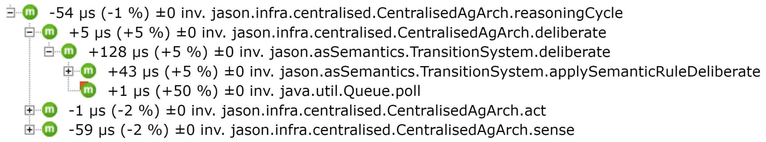
The observation that the idiomatic agent outperformed the goal-directed agent in the collision avoidance scenario, and the lack of performance from the reactive agent, raised a new question: Why was there a performance difference between these agents? To answer this question an experiment focusing on the performance of the reasoning cycle was performed to identify what was causing



(a) Crash rate. (b) Decision time.

Fig. 3: Agent Collision Avoidance Test Results.

this difference. This experiment used a standalone Jason project with a sequence of 50 perception sets passed to the agent for it to reason on. These perceptions were taken from the log files from the stop trial experiment and focused on a scenario where the agent should select a collision avoidance plan. Using JProfiler, the reasoning cycle was observed to find which aspect of the reasoning cycle was causing the difference in decision time. The profile results comparing the idiomatic agent to the goal-directed agent are provided in figure 4. This figure shows the difference in the call tree for the two agent runs, focusing on the differences in the average call times of agent's methods. As can be seen in this figure, the goal-directed agent spends significantly more time deliberating compared to the idiomatic version. More specifically, the agent spends an additional 43 μs in the `applySemanticRuleDeliberate()` method.



```
-54 µs (-1 %) ±0 inv. jason.infra.centralised.CentralisedAgArch.reasoningCycle
  +5 µs (+5 %) ±0 inv. jason.infra.centralised.CentralisedAgArch.deliberate
    +128 µs (+5 %) ±0 inv. jason.asSemantics.TransitionSystem.deliberate
      +43 µs (+5 %) ±0 inv. jason.asSemantics.TransitionSystem.applySemanticRuleDeliberate
      +1 µs (+50 %) ±0 inv. java.util.Queue.poll
  -1 µs (-2 %) ±0 inv. jason.infra.centralised.CentralisedAgArch.act
  -59 µs (-2 %) ±0 inv. jason.infra.centralised.CentralisedAgArch.sense
```

Fig. 4: Average Call Time Differences for Idiomatic and Goal-Directed.

With this result we can reassess why the goal-directed and reactive agents spend more time deliberating rules than the idiomatic agent. This was an interesting observation, as all three agents used the same rules, the only difference in these agents was when these rules were in scope for the reasoner. Table 4 summarizes how the plans that are triggered by the different events make use of the rules. In the case of the idiomatic agent, the obstacle avoidance behaviour used a single plan supported by a single rule. In the case of the goal-directed agent, the `waypoint` goal-triggered plans included four rules. In addition, the `waypoint` plans triggered sub-plans for controlling the speed and steering of the car which also used several rules. The reactive agent used a single event trigger which included plans that used all of the seven available rules.

Table 4: Agent Use of Rules.

Rule	Trigger That Uses Rule		
	Idiomatic	Goal-Directed	Reactive
obstacleStop	obstacle	waypoint	obstacle
atLocation	waypoint	waypoint	obstacle
nearLocation	waypoint	waypoint	obstacle
destinationBearing	waypoint	waypoint	obstacle
courseCorrection	controlSteering	controlSteering	obstacle
steeringSetting	controlSteering	controlSteering	obstacle
lkaSteering	controlSteering	controlSteering	obstacle

To understand the impact of these rules on the performance we must consider how a Jason agent selects which plan to set as an intention. When the agent starts deliberating it first selects which event to use and then selects which applicable plan triggered by the selected event will be set as the agent's intention. The reasoner only deliberates on rules that are used by plans triggered by the selected event. Rules that are not tied to the selected event are not processed. This means that in the collision avoidance scenario, the idiomatic agent only needed to deliberate on a single rule, and only had a single plan to choose from when setting its intention. By contrast, the goal-directed agent had four different rules that needed to be computed in order to select to most appropriate plan for performing collision avoidance. For the reactive agent, all seven rules needed to be processed for every reasoning cycle.

To summarize this finding, it seems that reducing the length of the context guards in plans and reducing the number of necessary rules can lead to a performance improvement at runtime. This finding is in line with the performance analysis in [18]. There seems to be a benefit of the separation of concerns between the plans: having each triggering event focus on a single issue with fewer plans needed for each triggering event. This can be helped by having the event selection function make appropriate decisions with respect to which triggering event should be selected. In the case of the idiomatic agent, this benefit was provided by the behaviour prioritization functionality.

6 Limitations and Future Work

A challenge faced in working on this paper was the limited guidance on different approaches that can be taken when designing AgentSpeak software. Also limited were metrics that should be used when evaluating the maintainability and complexity of AgentSpeak. For this reason, we proposed adapting the concepts of coupling, cohesion, and cyclomatic complexity. In the case of coupling and cohesion, these concepts were applied in *spirit*, without a formal definition. Various properties of the plans were used to assess the agents relative to each other. These properties were useful for this study but may not translate to other agent designs or other applications. Further work is needed to both formally define coupling and cohesion for AgentSpeak software as well as better align these metrics with other metrics used in the field of MAS, especially from the perspective of the design of individual agents.

The work in this paper was also limited to a single application domain – the driving of the AirSim car in a collision avoidance scenario. Additional tests in other domains would be useful for validating these results. Additional programming styles could also be tested, for example Jason's imperative style programming with conditional statements and loops. Comparisons with other agent programming languages, such as SARL or GOAL, could provide insight for how Jason and BDI agents perform in general.

7 Conclusion

This paper examined the trade-offs between the design of Jason BDI agents and the performance of these agents. Using a simulated car in a collision avoidance scenario, the performance of the agents was compared in terms of the duration of their reasoning cycles and their reaction to obstacles. The assessment of the design of the agents focused on software maintainability and complexity. Inspired by the object-oriented software community, the concepts of coupling, cohesion, and cyclomatic complexity were adapted for use with AgentSpeak code.

We found a performance benefit, especially in terms of the agent's responsiveness to stimuli, to designing AgentSpeak software with looser coupling and higher cohesion, which implied that the software was more maintainable. Unfortunately, this benefit was inversely related to the software's cyclomatic complexity, primarily due to the connections with other modules. Using profile testing, the cause of the performance difference was found to be the processing of rules – smaller plan contexts with fewer rules may lead to significant performance improvements. The reason for the performance difference being tied to the software properties is that having plans focused on fewer concerns resulted in less complicated plan contexts, which in this case meant less rule processing. The result translated directly to a performance advantage in terms of the time to react to obstacles as well as their relative coupling and cohesion. This suggests that well designed and more maintainable AgentSpeak code has a performance advantage over simpler but less maintainable AgentSpeak. Although the conclusion from this work could be interpreted to mean that rules may be causing a processing burden for the agent, the use of rules can be useful for reducing code duplication in the plan contexts. Therefore, good AgentSpeak code should make use of rules to help improve their properties of coupling and cohesion, despite the performance hit. Lastly, the BDI agents were easily outperformed by an imperative program written in Python which offered equivalent driving behaviour for the car, although this version did not come with the benefit of the reasoning cycle which only adopts goals it believes it can achieve, dropping goals that are no longer believed to be achievable or motivated.

Acknowledgement

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), [funding reference number 518212].

Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), [numéro de référence 518212].

References

1. Barnes, D.J., Kolling, M.: Objects First with Java: A Practical Introduction Using BlueJ. Pearson, Boston (2017)
2. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology). John Wiley &; Sons Ltd., The Atrium, Southern Gate, Chichester, West Sussex, England (2007)
3. Cardoso, R.C., Ferrando, A., Dennis, L.A., Fisher, M.: An interface for programming verifiable autonomous agents in ros. In: Bassiliades, N., Chalkiadakis, G., de Jonge, D. (eds.) Multi-Agent Systems and Agreement Technologies. pp. 191–205. Springer International Publishing, Cham (2020)
4. Far, B., Wanyama, T.: Metrics for agent-based software development. In: CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No.03CH37436). vol. 2, pp. 1297–1300 vol.2 (2003). https://doi.org/10.1109/CCECE.2003.1226137
5. Gavigan, P.: Agent in a Box Demo - Car Lane Keep and Obstacle Avoidance. https://youtu.be/tvqkNnpKIPo, accessed: 2021-04-05
6. Gavigan, P.: AirSim Navigating Car. https://github.com/NMAI-lab/AirSimNavigatingCar/, accessed: 2021-02-19
7. Gavigan, P.: Jason Car Agent - Crash Case. https://www.youtube.com/watch?v=vfc_YLg0X2I, accessed: 2022-04-08
8. Gavigan, P.: Jason Car Agent - Stop Case. https://www.youtube.com/watch?v=Rlp2wY3FDJU, accessed: 2022-04-08
9. Gavigan, P.: SAVI_ROS_BDI. https://github.com/NMAI-lab/savi_ros_bdi, accessed: 2020-02-18
10. Habiba, M.: Metrics for evaluating agent oriented software engineering model. In: 2012 International Conference on Informatics, Electronics Vision (ICIEV). pp. 17–22 (2012). https://doi.org/10.1109/ICIEV.2012.6317459
11. Hübner, J.F., Bordini, R.H.: Jason: a Java-based interpreter for an extended version of AgentSpeak. http://jason.sourceforge.net, accessed: 2019-02-16

12. Kinny, D., Georgeff, M., Rao, A.: A methodology and modelling technique for systems of bdi agents. In: Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Agents Breaking Away. p. 56–71. MAAMAW '96, Springer-Verlag, Berlin, Heidelberg (1996)
13. Kramer, S., Kaindl, H.: Coupling and cohesion metrics for knowledge-based systems using frames and rules. ACM Trans. Softw. Eng. Methodol. **13**(3), 332–358 (jul 2004). https://doi.org/10.1145/1027092.1027094, `https://doi.org/10.1145/1027092.1027094`
14. Lazarin, N.M., Pantoja, C.E.: A robotic-agent platform for embedding software agents using raspberry pi and arduino boards. 9th Software Agents, Environments and Applications School (2015)
15. McCabe, T.: A complexity measure. IEEE Transactions on Software Engineering **SE-2**(4), 308–320 (1976). https://doi.org/10.1109/TSE.1976.233837
16. Microsoft: AirSim. `https://github.com/Microsoft/AirSim`, accessed: 2019-03-27
17. Miles, S., Joy, M., Luck, M.: Designing agent-oriented systems by analysing agent interactions. In: Ciancarini, P., Wooldridge, M.J. (eds.) Agent-Oriented Software Engineering. pp. 171–183. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
18. Miller, J., Esfandiari, B.: Analysis of the execution time of the jason bdi reasoning cycle. In: Alechina, N., Baldoni, M., Logan, B. (eds.) Engineering Multi-Agent Systems. pp. 218–236. Springer International Publishing, Cham (2022)
19. Moores, T.T.: Applying complexity measures to rule-based prolog programs. Journal of Systems and Software **44**(1), 45–52 (1998). https://doi.org/https://doi.org/10.1016/S0164-1212(98)10042-0, `https://www.sciencedirect.com/science/article/pii/S0164121298100420`
20. Pantoja, C.E., Stabile, M.F., Lazarin, N.M., Sichman, J.S.: ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming. In: Baldoni, M., Müller, J.P., Nunes, I., Zalila-Wenkstern, R. (eds.) Engineering Multi-Agent Systems. pp. 136–155. Springer International Publishing, Cham (2016)
21. Serebrenik, A., Schrijvers, T., Demoen, B.: Improving prolog programs: Refactoring for prolog. In: ICLP (2004)
22. Shah, S., Dey, D., Lovett, C., Kapoor, A.: AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. In: Field and Service Robotics (2017), `https://arxiv.org/abs/1705.05065`
23. Soza, H.: Quality measures for agent-oriented software. In: Shikhin, V. (ed.) Multi-Agent Systems, chap. 2. IntechOpen, Rijeka (2019). https://doi.org/10.5772/intechopen.79741, `https://doi.org/10.5772/intechopen.79741`
24. Stabile, M.F., Sichman, J.S.: Evaluating Perception Filters in BDI Jason Agents. In: 2015 Brazilian Conference on Intelligent Systems (BRACIS). pp. 116–121 (Nov 2015). https://doi.org/10.1109/BRACIS.2015.18
25. Stevens, W.P., Myers, G.J., Constantine, L.L.: Structured design. IBM Systems Journal **13**(2), 115–139 (1974). https://doi.org/10.1147/sj.132.0115
26. Wesz, R.: Integrating Robot Control Into The AgentSpeak(L) Programming Language. Master's thesis, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil (2015)
27. Wooldridge, M., Jennings, N.R., Kinny, D.: The gaia methodology for agent-oriented analysis and design. Autonomous Agents and Multi-Agent Systems **3**(3), 285–312 (2000). https://doi.org/10.1023/A:1010071910869, `https://doi.org/10.1023/A:1010071910869`