Novice Programmers' Faults & Failures in GOAL Programs: Empirical Observations and Lessons

Michael Winikoff
Department of Information Science
University of Otago
Dunedin, New Zealand
michael.winikoff@otago.ac.nz

ABSTRACT

What are the types of mistakes ("faults") that novice GOAL programmers make, and how do they manifest as failures? This question is important since it has significant implications to the ongoing design of GOAL, and other agent-oriented programming languages; to the ongoing development of tools that support GOAL programmers; and to how we teach agent-oriented programming. In this paper we develop taxonomies for faults and for failures. We then classify the faults and failures that occur in a collection of programs by novice GOAL programmers. This provides empirical data which we use to make recommendations regarding the GOAL language, its support tools, and how it is taught.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software

General Terms

Measurement, Human Factors, Design

Keywords

Agent Oriented Programming, Faults, Failures, Taxonomy, GOAL

1. INTRODUCTION

What are the types of faults that novice programmers create when using Agent-Oriented Programming Languages (AOPLs) and how do they manifest as failures? This paper aims to answer this question. In doing so, we also address the related question of how do we *classify* faults and failures in agent programs? This work is significant in that having an understanding of what sorts of faults are created, and how they manifest as failures, has implications not just to how we teach agent-oriented programming, but also to how we design AOPLs and their associated development environments.

In this paper we focus on mistakes made by *novice* programmers using the GOAL agent-oriented programming language. Given the low number of software developers who are familiar with agent-oriented programming, if agent programming is to gain mainstream adoption, then we need to consider how to teach agent programming to many programmers who may be experienced in other forms of programming, but are still novices in agent programming. In other words, understanding how to make it easier for programmers to learn agent-oriented programming could contribute to easing the adoption of agent-oriented programming. This justifies the focus on novice programmers. The focus on GOAL is motivated by data availability. We return to this and consider how our work might generalise to other AOPLs in the final section of the paper.

Before going further, we need to define some terminology. Following standard usage (e.g. IEEE 610.12-1990) we use the terms *fault* and *failure*. A *fault* is a mistake in a program made by a programmer. A fault can result in a *failure* which is the runtime manifestation of the fault. In this paper we focus primarily on faults, but also on the failures.

It is worth noting that the relationship between faults and failures is not a simple one-to-one relationship. A single fault may manifest as multiple failures. Consider a GOAL rule of the form "*if* C *then insert*(ϕ) *and adopt*(ψ)". If the condition C is too weak, then this may result in incorrect updating of the agent's beliefs ("insert") *and* the incorrect adoption of goal ψ. This complex relationship between faults and failures means that we analyse them independently.

There has been very little work that has investigated agent-oriented software development empirically, and very little is known about how AOPLs are used in practice. Indeed, only recently van Riemsdijk *et al.* [11] called for more work in this area, arguing that in addition to using so-called "formulative" approaches in developing agent languages, the time has come to also adopt an *empirical* perspective.

This paper makes a number of contributions: it proposes a taxonomy for failures, a taxonomy for faults, provides empirical data on occurrence of faults and failures in GOAL programs, and, based on this data, considers implications for debugging tools, language design, and teaching agent programming to novices.

The remainder of this paper is structured as follows. We briefly review related work below, and then proceed (Section 2) to introduce GOAL, and the Blocks World for Teams (BW4T) problem. Our methodology is described in Section 3, and we then proceed (Section 4) to develop taxonomies for classifying faults and failures. The results of our analysis are presented in Section 5. We conclude the paper with a discussion of the implications of our findings, limitations, and future work (Section 6).

There has been surprisingly little work on debugging multi-agent systems, and even less on *empirical* investigation of bugs in agent programs. Earlier work on debugging MAS (e.g. [6]) focussed on tool support for debugging, rather than on understanding what faults and failures occur in agent programs. There has been a range of subsequent work that has looked at tool support for debugging — see [3] for a discussion of recent work in this area.

Appears in: *Alessio Lomuscio, Paul Scerri, Ana Bazzan, and Michael Huhns (eds.), Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), May 5-9, 2014, Paris, France.*

Considering work that seeks to understand what faults and failures occur in agent programs, Poutakidis *et al.* [9] is an early example of an empirical investigation of failures in agent programs. Given that their tool provides debugging support by eavesdropping on conversations and detecting deviations from the interaction protocol, their taxonomy, unsurprisingly, is focussed on failures that are exhibited through incorrect message behaviour (e.g. sending a message to the wrong recipient, or sending a message multiple times). They provide brief discussion, for each failure, of the faults that could give rise to that failure, but do not attempt to give a taxonomy of faults. Overall, the paper (which is short) does not provide details on the methodology, the derivation of the taxonomy, or the relative frequency with which different failure types occur.

More recent[1] is the work of van Riemsdijk *et al.* [11]. They considered a collection of GOAL programs written by programmers (some novice, some more experienced). However, although they adopted an empirical approach, their aim and methodology differ from ours. Whereas we are interested in understanding what faults and failures occur (which involves testing and debugging programs), they primarily studied and analysed the (static) program, looking at programming idioms, and (syntactic) use of different language features. Their first case study, a dynamic blocks world problem, involved three programs (two researchers and one programmer). Their second case study, Unreal Tournament 2004, involved 12 programs, each written by a team of first year students.

Even more recent is the work of Padgham *et al.* [7]. They assessed the effectiveness of a testing tool by applying it to 13 different student-written implementations of a CLIMA agent competition scenario (where agents roam a grid world looking for gold), and to a single implementation of a weather alerting system (developed by a research assistant). They proposed a taxonomy of faults, and explored how well their testing tool was able to find faults, and how many of these turned out to be real faults (as opposed to false positives). One difference is the aim of the work: they aimed to assess a testing tool, rather than to shed light on what faults occur. Another difference is that they consider the relationship between design and code, and consequently their fault model relates to inconsistencies between design and code (e.g. the event that triggers a plan in the code being different from the event that triggers that plan in the design).

Another area of related work is debugging (non-agent) parallel programs. However, although MAS can be seen as a special case of a parallel system, there are significant differences: e.g. parallel programming usually does not need to deal with a dynamic environment where things can, and often do, go wrong. Pedersen and Jones [8] examined what faults novices make when developing parallel programs. One interesting finding that they made was that most faults actually concerned sequential code. However, the minority of faults involving message passing were the ones that were harder to debug.

2. BACKGROUND

This section briefly introduces the GOAL programming language and the Blocks World for Teams (BW4T) problem.

2.1 GOAL

This section briefly introduces GOAL (**G**oal **O**riented **A**gent **L**anguage); for further details we refer the reader to the existing literature [1, 2]. A MAS in GOAL is specified with a configuration file (".mas2g") that provides configuration options, and specifies both the environment, and the GOAL agents to be started. Each GOAL agent is written in a file (".goal") that may include the following components. **Domain knowledge:** these are reasoning rules that allow the agent to draw conclusions by reasoning from its beliefs. In the current implementation, these are written in SWI-Prolog. **Initial beliefs** and **initial goals** that are created when the agent starts up. **Action definitions:** each action that the agent can perform in the environment is defined using pre and post conditions. **Rules:** A goal program can include a number of modules containing rules. In this paper we assume programs have two modules: a percept processing module ("event") and a main module.

The behaviour of the agent is specified using rules which are of the form[2] "**if** condition **then** action(s)". Each module (either percept or main) consists of a sequence of these rules. The condition of each rule is specified over the agent's beliefs and its goals. For example the (hypothetical) rule "**if** bel(in(Room), color(Block,Color)) , not(goal(deliver(ABlock))) **then** adopt(deliver(Block))" indicates that if the agent believes it is in a Room, and that it also believes that a Block has a certain Color, and doesn't already have a goal to deliver a block, then it should adopt a goal to deliver the Block. Each rule can contain one or more actions. Actions in GOAL are either user-defined, or are one of the five built-in actions that *insert* or *delete* beliefs, *adopt* or *drop* goals, or *send* a message. Where a rule has multiple actions, they are separated by "+" and are performed in order [2, Section 4.5], and there can be at most one user-defined action[3].

We now turn to the language's semantics. A rule is *applicable* if its condition holds, and is *enabled* if, additionally, the actions' preconditions are met. Applicable and enabled rules are *options*. The execution cycle consists of the following steps: (1) Clear previous cycle's percepts. (2) Update percepts by executing *all* options (i.e. enabled rules) in the distinguished `event` module. (3) Focus on the `main` module: compute the options, select one (by default rules are evaluated in linear order and the first option is selected), and perform its actions. (4) Update goals by dropping goals that are believed to hold.

Compared with other cognitive agent programming languages (e.g. Jason, JACK, Jadex, 2APL), GOAL's features are fairly typical (e.g. using beliefs and goals in rules to select a course of action). However, GOAL does have a number of distinctive features, the most relevant of which are: (a) The use of rules to process percepts; (b) The limitation that an action rule can only result in a sequence of actions, rather than a mixture of actions and subgoals, and that only a single user-defined action can be included; and (c) The lack of a trigger in rules. This makes GOAL action rules more general, in that a rule doesn't require a particular trigger. However, it also means that a rule can be applied repeatedly: in, say, Jason a rule of the form $+!goal : context \leftarrow planBody$ can be applied (if the *context* is true) to deal with the creation of a *goal*. However, the rule will not be applied subsequently unless the goal is re-posted. By contrast, in GOAL a rule of the form "if goal(*goal*) then *actions*" can be applied repeatedly, as long as *goal* remains a goal of the agent. This repeated application means that GOAL programs are prone to infinite loops: if a rule is applicable, and its actions do not change the state, then (unless percepts change the agent's state), the rule will also be applicable in the next cycle, and the next, etc.

[1]The 2012 journal paper brings together and extends their earlier work, going back to a 2009 PRIMA paper.

[2]There is also a form "**forall** MSC **do** actions" used in the percept processing module.

[3]"The + operator can be used to combine as many actions as needed into a single, complex action, **as long as it respects the requirement that at most one user-defined action is included** in the list of actions combined by +." [2, Section 3.2, emphasis added]

2.2 Blocks World for Teams

The programs we examined each implement a solution to "Blocks World for Teams" (BW4T): a single agent that moves around an environment with a number of rooms, collecting blocks of various colours, and delivering them to the "dropzone" in a specified order (e.g. a red block, then a blue block). The environment (which runs in a separate process) provides the agent with percepts (e.g. in(Room), color(BlockID, Color), holding(BlockID)), and four actions (goTo(Location), goToBlock(BlockID), pickUp, and putDown).

The challenge is to develop an agent that explores the environment, delivering blocks when it can, and that is also able to deal with opportunities (e.g. finding itself beside a block that is of the right colour), without getting "distracted" (e.g. finding itself beside a block that is *not* the right colour).

3. METHODOLOGY

In order to shed light on the faults that are present in agent programs, and the associated failures, we need to consider a collection of such programs, identify the failures and faults, and classify them using appropriate taxonomies. We obtained a collection of 55 GOAL programs for the BW4T problem that were written as an assignment by first year undergraduate students at Delft university.

The students were provided with a skeletal program (which is included with the GOAL distribution). The program includes a very small amount of existing code: domain knowledge rules defining the predicate `room(X)` (two lines), an initial belief `state(unknown)` (1 line) and the definition of the action `goTo(Location)` with pre-condition `state(arrived) ; state(collided)` and a true post-condition (3 lines). The skeleton program also contains two rules (2 lines) in the main module that adopt goals to explore, and go to places; and rules (total 6 lines) in the percept processing module that deal with `at(Place)` and `state(State)` percepts. Finally, the program also contains comments that prompt the student to complete different parts of the program, for instance the action specification part prompts the student to insert specifications for `goToBlock`, `pickUp` and `putDown`; the domain knowledge has a comment prompting the student to define a predicate `nextColorInSeq(Color)`, and the main module prompts the student to "*improve the two lines of code above such that the agent checks the rooms in a more efficient way (not checking the same room twice)*" and to "*insert code that lets the agent deliver a block when it knows about a block that can be delivered. Make use of the goals: delivered(Pos), in(Room) and atBlock(Block)*".

We then used the following process (see Figure 1).

1. **Test**: Each program was tested thoroughly to find bugs. Test adequacy is a potential issue, and so when testing the programs we used a number of test suites (with over 7300 tests): the two example scenarios that were used in the original assignment, a set of ten randomly generated test cases, and a generated enumeration of all possible starting configurations within a limited scope[4], which generated 729 starting configurations for scope $N = 2$ and 6561 for $N = 3$. We considered a program to be correct if it managed to deliver the desired blocks in all runs. However, we did not require that programs did this efficiently: in some cases programs also delivered additional, incorrect, blocks, but still delivered all required blocks, and were therefore considered correct.

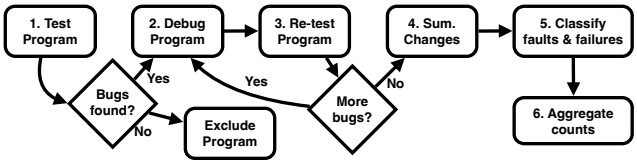

Figure 1: Methodology

If no bugs were found, the (correct) program was excluded from consideration. Otherwise we continued to the next step. Of the 55 programs, 4 could not be run at all and were excluded[5], and 10 had no bugs and were also excluded, leaving 41 buggy programs. To give a rough sense of their size, the 51 programs (41 buggy plus 10 with no bugs) ranged from 172 lines (as counted by `wc`) to 378 lines, with a mean of 225.49 lines and a median of 220 lines.

2. **Debug**: We fixed the bugs (which was very time consuming!), taking care to consider possible alternative fixes, and to make as few changes as possible (see below).

3. **Re-test**: We then re-ran the tests to ensure that we had in fact fixed the program. If bugs still existed, then we returned to the previous step.

4. **Summarise changes** in the program. We compared the original and corrected programs (using `diff`) to identify all differences. Each difference between the original and corrected program is a fault.

5. **Classify faults & failures**: Each fault in the program, and associated failure, was classified using the taxonomies presented in Section 4. In assigning a taxonomy category to failures we ignored the actual observed behaviour of the program, and instead considered the fault in isolation and asked: "what failure would be exhibited if this fault were the *only* fault in the program?". This was done because the relationship between failures and faults is complex and the actual observed behaviour of a program may be due to a combination of faults. See Section 4.3 for an example of classification.

6. **Aggregate counts**: We counted how many programs exhibited faults (resp. failures) in each category of the taxonomy.

In applying this methodology, there were a number of issues that had to be considered and resolved. Firstly, when debugging programs (step 2), there are obviously a number of different ways of fixing a given program. We carefully considered alternative fixes, and, where there were different options, selected the simpler option. Where there were multiple possible fixes that were equally simple we preferred to make changes to the detected source of the fault, rather than elsewhere in the program (since this is arguably the likely debugging behaviour), and we considered the place in the program where an issue was first detected, rather than digging deeper to find root causes. We were also careful to only make necessary changes (and, if in doubt, only made those changes that were clearly essential, and then ran additional tests to confirm whether further changes were needed). For programs that were very close to being correct (few changes) it turned out to be easy to debug them

[4]We considered all possible arrangements of $N + 1$ blocks in the environment (possible colours drawn from red, blue, green; possible locations RoomA1-RoomA3) where N blocks had to be delivered.

[5]Since the programs weren't runnable, they would not have been tested or debugged. We considered these un-tested and undebugged programs to not be appropriate for inclusion, since they were likely to be immature and to contain many bugs.

and to identify necessary changes (see Section 4.3 for an example). On the other hand, for programs that were very buggy, there was more freedom in how the program could be fixed, and so we excluded such programs. Specifically, five of the 41 buggy programs required a large number of changes[6] (> 10), and were excluded. **This left 36 programs that were used in our analysis.** Most of the programs considered (75%) required 6 or fewer changes.

An issue that arises when classifying faults (step 5) is that we cannot reliably ascertain the programmer's intention. This means that we cannot distinguish between fault types where the distinction is based on the programmer's intentions. We mitigated this issue by having taxonomies that are fairly "coarse grained", and avoiding intention-based categories (see Section 4).

A final issue is how to count multiple instances of the same fault type (step 6). For example, suppose a given program has two different percept processing rules that have conditions that are too strong, and as a result, fail to deal with (different) percepts. Should we count this fault type twice, or just once? We resolved this issue by reporting results in terms of the number of *programs* that exhibited a particular fault type, or failure type, rather than counting the number of fault (or failure) types. This means that it didn't matter whether a program exhibited the same failure (or fault) type multiple times: it still counted as one program that had that fault type. Given that our aim is to assess how likely certain faults are, this approach makes sense: it tells us, across different students, how many students create certain fault types.

4. TAXONOMIES

In developing taxonomies for failures and for faults the first question to consider is whether we should develop our taxonomies based solely on the observed failures and faults ("bottom-up"), or whether we should develop taxonomies "top-down" based on some principles, and then extend them based on the observed failures and faults. We argue for using a top-down approach for two reasons. Firstly, top-down analysis can suggest possible types of failures or faults that might be expected to occur, and it is then possible to detect a situation where a certain fault or failure type can occur in principle, but in practice is not observed. This sort of finding is of interest, but is impossible to make with a solely bottom-up taxonomy. Secondly, a bottom-up taxonomy has a tendency to be somewhat unstructured and to mix levels of abstraction. For example, Pedersen and Jones' [8] (bottom up) taxonomy has 10 categories of faults including such things as memory-related faults (e.g. pointers, arrays), faulty hardware, and redefining system keywords.

However, we do not use a pure top-down approach: the taxonomies are modified in light of the observed faults and failures. For example, when classifying faults, we observed faults where "if then" was used instead of "forall do" in the percept module. We therefore extended our fault taxonomy with this new specific type of fault, based on empirical observation. The list of fault types (Figure 2) includes an indication ("EO") that flags the types that were added based on empirical observation.

So, what principles can we use to develop taxonomies? One option for classifying faults is to consider where they were introduced in the programming process. However, this does not apply to failures (which are only exhibited when an implementation is run), and requires an *observational* study (such as [5]): given that we only have the final submitted program, we cannot reliably distinguish between requirements issues (programmer doesn't understand requirements), design issues, and implementation issues.

[6]Where we define a "change" as a single simple syntactic change, specifically the application of one of the mutation rules of [10].

Another possible basis for developing taxonomies is *language features*. Indeed, this appears to be a common approach (e.g. see Table 1 in Ko & Myers [5]). However, there is considerable variance in the level of categories in the taxonomy. Some taxonomies classify bugs based on quite specific language features (e.g. bugs involving assignment statements vs. bugs involving arrays). Other taxonomies focus more on higher-level concepts, sometimes quite broad (e.g. "blunder or botch" [4], described as "*...I knew what I ought to do, but I wrote something else that was syntactically correct—sort of a mental typo*").

We argue that for both taxonomies (faults and failures) it makes sense to base the taxonomy on the language, but that the two taxonomies should be based on different aspects: the fault taxonomy is based on the *syntactic* structure and features, whereas the failure taxonomy is based on the language's *semantics*. Recall that a fault is a mistake that the programmer has made in the program, and a failure is the *runtime manifestation* of this fault.

For developing a top-down taxonomy of faults we consider the syntactical structure and features of the GOAL language. This is because faults are in the program, and so the language's structure and features are, in essence, the possible places where faults can exist. However, as discussed in the previous section, we want a fairly high-level taxonomy, in order to make it easy to assign categories to faults. We therefore begin not with specific language features of GOAL, but with the top-level structure of a GOAL program.

On the other hand, for developing a top-down taxonomy of *failures*, we consider the *semantics* of the language. Failures are exhibited at runtime, and so it makes sense to classify failures in terms of the execution of the language, i.e. its semantics.

Note that the location of faults and the resulting type of failure are expected to correlate. For example, a failure involving percept processing would usually be the result of a fault in the percept module. However, there are situations where this will not be the case. Real examples: an incorrect action may be selected due to a fault in the knowledge-base which leads to a condition in a rule being incorrectly evaluated to false; or an incorrect action may be selected because the correct action is not applicable (enabled), due to a fault in the action definition part of the agent.

4.1 Fault Taxonomy

We derive our fault taxonomy by considering first the overall structure of a GOAL program, and secondly, the detailed syntactical features of GOAL. Recall (Section 2.1) that a GOAL program consists of: action definitions (defining the interface between the MAS and the environment), domain knowledge, initial beliefs and goals, and percept processing and action selection rules. Note that we did not consider the configuration file to be a likely source of faults, since it is simple and fairly standard across projects, and since any faults in the configuration would prevent the system from running, and hence be easily detected and quickly fixed.

Considering a list of rules, each of the form "if *condition* then *action(s)*", what sorts of faults can occur? Obviously, it is possible for a rule to be missing (case (a) in Figure 2), or for a completely new (and incorrect) rule to be present (b). It is also possible for the *condition* of the rule to be erroneous (c) or for the *action(s)* of the rule to be wrong in some way (d). We also observed early on that the *order* of rules is a rich source of faults: in the current version of GOAL rules are considered (by default) in linear order, with the first enabled rule being selected. This means that if more than one rule is enabled, then putting the desired one later, may mean that another earlier (enabled) rule is erroneously selected (f).

Considering the initial beliefs and goals, we define a fault class for these (n). We do not refine this further. We could have defined

Fault Taxonomy

a:	missing rule
b:	additional (wrong) rule
c:	condition on rule wrong - specific variants:
	cs: condition too strong (EO)
	cw: condition too weak (EO)
d:	action(s) of rule wrong (but legal)
e:	rule includes two user-defined actions
f:	rule order wrong (EO)
g:	action definition wrong
i:	using "if then" instead of "forall do" (EO)
j:	missing action in a rule (special case of "d", EO)
k:	fault in domain knowledge
n:	fault in initial beliefs/goals
t:	typo (e.g. `atblock` instead of `atBlock`)
o:	other fault not classified above

Failure Taxonomy

P1:	failure to deal with percept
P2:	other incorrect percept processing
G1:	failure to add goal that should be added
G2:	failure to drop goal that should be dropped
G3:	adding a goal that shouldn't be added
G4:	incorrectly adding a second goal of the same type (special case of G3, EO)
G5:	dropping a goal that shouldn't be dropped
A1:	selecting wrong (user-defined) action
A2:	beliefs not updated correctly when action performed
A3:	action selected when should be doing nothing (waiting for environment, EO)
A4:	action interface mismatch (EO)
O:	other failure not classified above

Figure 2: Fault Taxonomy (left) and Failure Taxonomy (right). "EO" denotes Empirical Observations.

separate categories for faults in initial goals, and faults in initial beliefs. However, the programs did not actually use initial goals.

Considering the domain knowledge, we again chose to define a single broad category for faults involving domain knowledge (k). We note that the programs did not make extensive use of domain knowledge rules.

Considering action definitions we also define a single broad fault class for these (g). Finally, we also define two general types of faults: typos (t), and an "other" (o) category to cover anything else. Note that the choice of labels (e.g. "a") is intended to be somewhat mnemonic, e.g. using "i" for "using if-then . . . ", and "n" for faults with i**N**itial beliefs/goals.

We subsequently extended the fault taxonomy based on empirical observation (denoted "EO" in Figure 2). Firstly, it turns out to be useful to distinguish two specific sub-cases of a condition in a rule being wrong: where a condition is too *strong* (cs), and where it is too *weak* (cw). Another observation was that a number of programs had rules that were technically illegal, since they had more than one user-defined action in their action part. However, the GOAL implementation appears to accept them, but they do not behave as one might expect. We therefore defined a specific category for "action part wrong because it has more than one user defined action" (e), and added a "(but legal)" condition to (d). We also observed that a specific case of the action(s) in a rule being wrong is for there to be a missing action (j). Finally, we observed that using "if then" instead of "forall do" is a possible fault in the percept module (i).

4.2 Failure Taxonomy

We derive our failure taxonomy by considering the execution cycle (i.e. semantics) of GOAL. Recall that the execution cycle of GOAL distinguishes between percept processing and applying actions. This suggests that we distinguish between failures involving percept processing, and failures involving actions. Furthermore, GOAL has three types of actions: user-defined (which take place in the environment), belief update, and goal update. This, together with the existence of a step in the execution cycle that checks for achieved goals and drops them, suggests that we should consider distinguishing between failures that relate to user-defined actions, those that relate to belief updates, and those that relate to goal maintenance. We therefore consider three main types of failures: **P**ercepts, **G**oals, and **A**ctions (covering both user-defined and belief update actions).

For each of these three top-level aspects (P, G, A) we consider what possibilities exist for incorrect behaviour. In general one can fail to do what one should have done (e.g. not deal with a percept, or fail to drop a goal when it should be dropped), or one can do something that should not have been done (e.g. dropping a goal that should have been kept).

Considering percepts, we define two failure types: failure to deal with a percept (P1), and dealing with a percept incorrectly (P2). The first (P1) can be seen as a special case of the second where a percept arrives and does not result in any change.

Considering goals, we could fail to add a goal that should be added (G1), fail to drop a goal that should be dropped (G2), add a goal that should not be added (G3), and drop a goal that should not be dropped (G5). In addition, we observed, when debugging programs, that a common special case of G3 is adding multiple instances of a given goal "type" (G4), e.g. having both a goal to be `atBlock(12)` and `atBlock(14)`, which results from adding the second goal when there already is a goal of that type.

Considering action selection, one failure type is selecting the wrong user-defined action (A1), which includes not having an action that can be performed[7]. Another failure type is updating beliefs incorrectly when an action is performed (A2), which can be due to incorrect action definition (post conditions) or to missing/incorrect belief update actions in the rule. Another failure type is selecting an action to perform when we should be doing nothing (A3). This failure type is only applicable when the environment executes asynchronously (as it does in BW4T), and therefore there may be situations where the agent should be doing nothing while waiting for actions to complete. Finally, another failure type relating to actions is attempting to perform non-existent actions (A4), e.g. doing `putDown(`*`blockID`*`)` instead of `putDown`).

The fault and failure taxonomies are summarised in Figure 2. In order to avoid confusion between failure types and fault types we use lower case letters ("a", "b", "c") to refer to fault types, and upper case letters (and numbers) to refer to failure types ("P1", "P2").

[7]We originally intended to distinguish between selecting the wrong action and not having any action that could be performed. However, it turned out to be difficult to clearly distinguish these two cases.

4.3 Example

We now briefly illustrate the process of debugging and classifying faults and failures for one example program. Space precludes presenting the full program. One of the programs that we considered exhibited the following behaviour: ... goTo(RoomA1), goToBlock(44), pickUp, goTo(DropZone), putDown, goTo(RoomA1), goToBlock(44), goToBlock(44), ...

Debugging eventually traced the fault down to the following line of code: "if bel(in(Room), nextColorInSeq(Color), color(Block, Color), not(holding(_)), pos(Block, Room)) then adopt(atBlock(Block))". The aim of this rule is to adopt the goal to reach an appropriately coloured block. The conditions on this rule indicate that the agent believes it is in a Room, that the next block to be delivered should be of a given Color, that it believes there exists a Block with that Color, which is positioned ("pos") in the current Room, and that the agent is not holding anything.

However, what was happening was that the agent was adopting a goal to reach a block that had already been delivered to the drop zone, and didn't exist any more. This meant that the action `goToBlock` could not succeed, and had no effect, leading to an infinite loop. The agent program in question already kept track of which blocks had been delivered using a `gone(blockID)` belief, so the simplest fix was to add `not(gone(Block))` to the condition of this rule. An alternative fix would have been to modify another part of the program to delete `pos(Block,Room)` when a block is picked up, but this involves another part of the program, and so is arguably not the natural fix that a (novice) programmer would consider.

We classify the fault as being a too-weak condition (cw). The failure is classified as being of type G3, adding a goal that should not be added.

5. ANALYSIS AND RESULTS

Having presented the taxonomy, we now present the results of analysing faults and failures in the GOAL student assignments. Recall that the overall question we are seeking to answer is "What are the types of faults that novice GOAL programmers make?", we consider both the faults they make and how these manifest (failures). However, our focus will be primarily on the fault analysis, since that is what we are most interested in: failures are secondary (the manifestations of faults).

We followed the methodology discussed in Section 3. Note that a single program can have multiple failures (or faults) in the same category, e.g. one particular program might have two faults: a rule that is in the wrong order (f), manifesting as an incorrect selection failure (A1), and a rule that has a too-strong condition (cs), also manifesting as incorrect action selection (A1). This program will count once under category "f" and once under category "cs", but, in the failure aggregation, will only count once (under "A1").

The remainder of this section presents an analysis of the occurrences of faults (5.1), and of failures (5.2). We highlight significant observations along the way and discuss their implications.

5.1 Faults

The following table shows the number of programs that contained one or more faults of a given type, for each type. For example, the entry "a 5" indicates that 5 of the 36 programs had at least one fault where a rule was missing. The left side of the table is sorted by category, the right side by the count (highest to lowest).

Fault	Count	Fault	Count
a	5	f	19
b	3	cw	15
c	5	e	9
cw	15	j	8
cs	7	cs	7
d	2	g	6
e	9	a	5
f	19	c	5
g	6	o	3
i	2	b	3
j	8	d	2
k	0(*)	i	2
t	2	t	2
o	3	k	0(*)
n	0	n	0

(*) See discussion in text later in this section.

OBSERVATION: Almost all fault types occurred in at least one program. However, of the 15 different fault types, only 8 occurred in more than 10% of the 36 programs, and only 4 types (f, cw, e, j) occurred in more than 20% of the programs.

The most common fault type was incorrect ordering of rules (f).

The finding highlights the importance of rule order to programming and debugging GOAL. Implications to debugging tools: perhaps tools could check for overlap between rule conditions, and warn of situations where multiple rules are applicable. Implications to teaching and programming practice: we could recommend that programs be written in a way that does not depend on rule order (by including additional conditions). Students could also be instructed to carefully consider the order of rules when programming and debugging.

The next most common fault type had to do with conditions. Note that if we consider "cw" and "cs" as being special cases of "c", then the number of programs that would have a "merged" "c" code is 20 (7 programs have two of c, cw, cs), which would make "incorrect condition of some sort (including too-weak and too-strong)" the most common fault type (ahead of rule order faults).

Faults with rule conditions were common. Faults where conditions were too weak (cw) occurred in twice as many programs as faults where rule conditions were too strong (cs).

It is worth noting that issues with order are actually related to issues with conditions: if conditions are written in such a way that there is only ever a single applicable rule, then the order doesn't matter. We further observe that of the 36 programs, 31 had at least one fault relating to conditions (i.e. in either category "f" or the merged category "c"), and that 14 of the programs had *only* faults that related to conditions. We thus conclude that:

Faults relating to conditions, directly or indirectly via rule order, are much more common than other fault types.

The next two most common fault types both relate to actions. The first, which is the third most common fault type, is using GOAL in a way that is technically illegal, but is accepted by the interpreter.

Attempting to use more than one user-defined action in a rule is not uncommon ("e", 9 programs)

This is somewhat surprising: we did not expect illegal GOAL goal programs to be a common type of fault. However, it does highlight that there is a tendency by (novice) programmers to want to perform a sequence of actions in response to certain situations. This is a feature that is common in other AOPLs, but not available in GOAL. This suggests that GOAL either be extended to deal with such rules appropriately (making it closer in semantics to other AOPLs), or that this situation, which technically is not allowed in GOAL, be detected and the user given an error message, or at least a warning. The fourth most common fault type (j) is a missing action.

We observed earlier that faults relating to conditions were common. If we consider faults relating to rules, which encompass both condition-related faults, and certain action-related faults, then this covers the vast majority of fault types found.

Faults that were not related to rules were very rare.

There are three types of faults that do not relate to rules (g, n, k). The most common type involved action definitions (g, 6 programs). It is worth noting that this may be somewhat high due to the BW4T environment running as a separate parallel process. This means that when an action is performed, the correct post-condition is "true": the expected effects of the action actually occur later, as a result of percepts being received from the environment. There were no programs with initialisation-related faults (n), and only one program had a bug in the knowledge-base (k) which led to a problem in applying rules (specifically goal adoption). However, in fact the fault was likely a typo: ">" was used instead of ">=", and so was classified as a typo (t) rather than a domain knowledge issue (indicated by "0(*)" in the table). The other typo fault was a program that in one place had `atblock` instead of `atBlock`.

Typos very rare: only two cases, and one of these may have been a conceptual fault ($>$ vs. $\geq$)

This is an interesting observation: although GOAL is *not* statically typed, typos appear to be rare in student debugged programs, even when those programs still contained bugs.

Finally, there were only three programs that had faults that did not fit into the taxonomy (i.e. were classified as "other"). One involved using "forall" instead of "if" (i.e. the opposite of class "i"). Another had issues with variable binding and scope due to nested rules. The third program had a percept rule that used "at(Block)" instead of "atBlock(Block)".

5.2 Failures

We begin by considering the most abstract categories: goals, percepts, and actions. In what aspect of execution were failures manifested? Of the 36 programs, 17 have a fault that manifested in relation to percept processing ("P", 47%), 17 have a goal-related failure ("G", 47%), and 31 have a failure manifesting in relation to action selection ("A", 86%). All of the failures were classifiable using our taxonomy (i.e. there were no failures classified as "Other").

More than half of the programs had no failure that manifested in relation to percept processing. More than half of the programs had no failures relating to goal handling. The vast majority of programs had failures with selecting the appropriate action.

Although action-related failures were the most common, it is worth highlighting that percept-related failures were not uncommon. Therefore, the percept module should not be ignored as a potential source of failures, which has implications to making sure that debugging tools support debugging percept processing (we provide specific suggestions in the next section).

Failure	Count	Failure	Count
P1	12	A1	29
P2	10	P1	12
A1	29	P2	10
A2	10	A2	10
A3	2	G3	7
A4	1	G4	7
G1	5	G1	5
G2	4	G2	4
G4	7	A3	2
G3	7	A4	1
G5	1	G5	1
O	0	O	0

We now briefly consider more specifically the type of failure (e.g. "A1" vs. "A2", etc.). There are only two types of percept-related failures, and they are roughly equally common. Considering actions, wrong action selection (A1) is the most common form of failure, but incorrect updating of beliefs (A2) is not uncommon. On the other hand, interface mismatch (A4) and failing to do nothing (A3) were rare. Interface mismatch faults (A4) may be somewhat low because students were provided with an example definition of the `goTo` action, and prompted to define the other three actions. Both cases of failing to wait for the environment (A3) were caused by the action precondition not including a test for the agent having finished travelling. Regarding goals, one interesting observations is that the three most common goal-related failures had to do with *adopting* goals (G1, G3, G4), whereas failures relating to *dropping* goals were less common (G2, G5). This may be a reflection of the fact that all goals have to be explicitly adopted, but in some programs dropping goals is done automatically when the goal is believed to be achieved.

6. DISCUSSION

We have developed taxonomies for classifying faults and failures in GOAL programs, and empirically explored the occurrence of faults and failures in a collection of programs written by novice GOAL programmers. Our observations have a number of implications for the ongoing design and development of the GOAL language, its support tools, and for how GOAL, and more generally agent-oriented programming, are taught.

Implications for language design: Firstly, given our observation that the percept processing module is a significant source of faults, it may be worth considering providing the programmer with simpler ways of specifying percept processing. For example, when a percept is received, the user often needs to have two rules: one to deal with the case where a belief already exists and needs to be updated, and another to deal with the situation where there is no existing belief. One possible extension to the GOAL language might be an abbreviation that defines that a percept maps to a belief of the same name, and then an appropriate collection of rules could be generated to ensure that the beliefs are correctly updated when a percept is received. Two specific cases are where there should only be at most one belief at a time (e.g. location of agent), and where multiple beliefs are possible (e.g. known locations of blocks). For other AOPLs, a recommendation is to consider (if relevant) how percept processing is specified.

A second implication to language design, which is specific to GOAL, concerns rules that have more than one user-defined action. This is technically illegal in GOAL, but the interpreter accepts such rules. In other AOPLs a rule with actions $A_1 + A_2$ is interpreted as "do A_1, and when it has completed, do A_2". However, GOAL attempts to perform both actions in the same cycle, which typically does not give the behaviour that the programmer expects. One option is to extend GOAL with multiple-action rules that would bring it closer to other AOPLs. Another option is to reject a rule with more than one user-defined action (or at least give a warning).

Implications for teaching: There are a number of faults that can be avoided relatively easily by adopting certain good programming practices such as: (1) Don't drop goals explicitly using the `drop` action, instead, define goals that match beliefs and allow GOAL to drop them automatically (this avoids issues with dropping goals incorrectly); (2) Aim to define conditions so that in each given situation only a single rule is applicable. These recommendations are also applicable to other AOPLs, since many of them provide an explicit "drop goal" operation, and use multiple rules with conditions to select between them.

Implications for tool design: The finding that fault types are dominated by condition-related faults, including rule order, is actually a positive one, since it is possible to check rules for overlap between their conditions, and provide warnings about situations where multiple rules apply, prompting the programmer to consider whether the rule order that they have specified is correct. This advice is also relevant to other AOPLs that use multiple rules with conditions. A second implication is that the debugging tool should provide support for debugging percept processing issues. It would be desirable to have a way of comparing percepts with belief updates, and perhaps even automatically raise a warning if percepts don't result in any belief changes, or, more precisely, if each individual percept does not result in some change to the agent's beliefs or goals.

Our work does have a number of issues relating to validity, both internal and external. Regarding internal validity, we only considered one problem (BW4T) that involves a single, relatively simple, agent, although we did consider quite a few programs. It is possible that the patterns of faults and failures are specific to the BW4T problem. We noted one place (action definitions) where this is likely the case. We also only looked at bugs in final submissions, which may mean that easier-to-fix bugs are not present. However, even if that is the case, this is arguably a good thing, since we are interested in those bugs that are harder to find and fix.

Regarding external validity, we only considered GOAL programs, and given that GOAL does differ in some significant ways from other AOPLs, it is not clear to what extent our findings generalise to other AOPLs. Clearly some findings (e.g. using more than one user-defined action) are specific to GOAL. However, other findings, such as the significant number of condition-related faults, or the importance of rule order, may also apply for other AOPLs.

Considering the taxonomies that we have defined, we expect the fault and failure taxonomies to require some changes to be applicable with other AOPLs. However, we expect these changes to be relatively minor, and this expectation is supported by a comparison of our fault taxonomy with the (independently developed) corresponding taxonomy of Padgham *et al.* [7].

Our work is complementary to van Riemsdijk et al's [11] analysis of how GOAL was used. They did not study faults and failures, but rather, considered programming style, and usage of features of the language. We note that they had a number of observations that relate to our findings, and support them. Firstly, they noted that single instance goals were used (which relates to our failure type G4). Our finding that failures relating to multiple instances of a goal type are not uncommon (7 programs) supports the suggestion to provide specific language support for single instance goals. They also observed the use of "durative" actions, which we have also highlighted as a source of faults (g).

There is a range of future work that would be worth pursuing, including looking at more problems, including systems with multiple agents, looking at other AOPLs, and developing a more fine-grained taxonomy for condition-related faults. There is also future work in using these results to improve AOPLs and their support tools.

7. ACKNOWLEDGMENTS

I would like to thank Koen Hindriks and Maaike Harbers from Delft University of Technology for providing the programs, and for answering questions about the BW4T environment implementation, and about GOAL. I would like to thank Sharmila Savarimuthu for her analysis of bugs in the context of mutation testing [10], which provided data that was the basis for this paper's analysis.

8. REFERENCES

[1] K. V. Hindriks. Programming rational agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*, chapter 4, pages 119–157. Springer, 2009.

[2] K. V. Hindriks. Programming rational agents in GOAL. Available from http://mmi.tudelft.nl/trac/goal, May 2011.

[3] K. V. Hindriks. Debugging is explaining. In *PRIMA 2012: Principles and Practice of Multi-Agent Systems*, pages 31–45. Springer, LNCS 7455, 2012.

[4] D. Knuth. The Errors of TEX. *Software—Practice and Experience*, 19(7):607–685, July 1989.

[5] A. J. Ko and B. A. Myers. Development and evaluation of a model of programming errors. Human-Computer Interaction Institute Paper 184, http://repository.cmu.edu/hcii/184, Carnegie Mellon University, 2003.

[6] D. T. Ndumu, H. S. Nwana, L. C. Lee, and J. C. Collis. Visualising and debugging distributed multi-agent systems. In *Proceedings of the third annual conference on Autonomous Agents*, pages 326–333. ACM, 1999.

[7] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller. Model-based test oracle generation for automated unit testing of agent systems. *IEEE Transactions on Software Engineering*, 39(9):1230–1244, 2013.

[8] J. B. Pedersen and M. Jones. Error classifications for parallel message passing programs: A case study. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 387–394, 2012.

[9] D. Poutakidis, L. Padgham, and M. Winikoff. An exploration of bugs and debugging in multi-agent systems. In *Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 628–632. Springer, LNCS 2871, 2003.

[10] S. Savarimuthu and M. Winikoff. Mutation operators for the GOAL agent language. In *Engineering Multi-Agent Systems (EMAS) post-proceedings*, LNCS, 8245, pages 255–273. Springer, 2013.

[11] M. B. van Riemsdijk, K. V. Hindriks, and C. M. Jonker. An empirical study of cognitive agent programs. *Multiagent and Grid Systems*, 8(2):187–222, 2012.