Model-Based Diagnosis and Execution on Rovers

Lars Blackmore
Steve Block
Thomas Léauté
Emily Fox

5th December 2003

1 Introduction

Our team has been working on the use of Titan to automatically estimate the state of a rover and recover from potential failures. The behavior of a rover is not only dependent on the behavior of its internal components; it is also affected by the environment in which it is evolving.The onboard computer running Titan should be able to detect failure modes related to non-nominal interactions with the environment, diagnose them (and hence discriminate them from internal component failures), and try to recover from them. We have been focusing on the development of a model for the rover that would enable us to diagnose and reconfigure the rover, subject to "environmental" failures (slipping and sliding) as well as internal component failures.

After a brief introduction to Titan and the rovers (section 2), we will describe the overall scenario we chose, which has been the goal and guideline of our project (section 3). We will then describe the four major model design steps we took, each step solving issues raised by the previous step and raising new issues due to the increasing complexity of the model (sections 4 to 7). We will then present in detail the results we got from the final model, tackling the initial scenario goal and even expanding the possibilities of the model to several other complex scenarios (section 8). We will eventually sum up the lessons we learned from the successive modeling issues we had to face, along with the strengths and weaknesses of our final model and of the tools we have been using (section 9).

2 Overview of Technology

2.1 Titan

Titan is a model-based executive controller that was used to estimate the state of the rover, diagnose possible failures from the observations output by the sensors onboard the rover, and suggest appropriate actions in order to recover from those failures when possible.

Titan uses a plant model of the rover in order to account for the behavior of each of the internal components and their interactions one with each other. This model can be represented by a combination of non-deterministic finite automata, an example of which is shown in Figure 1.

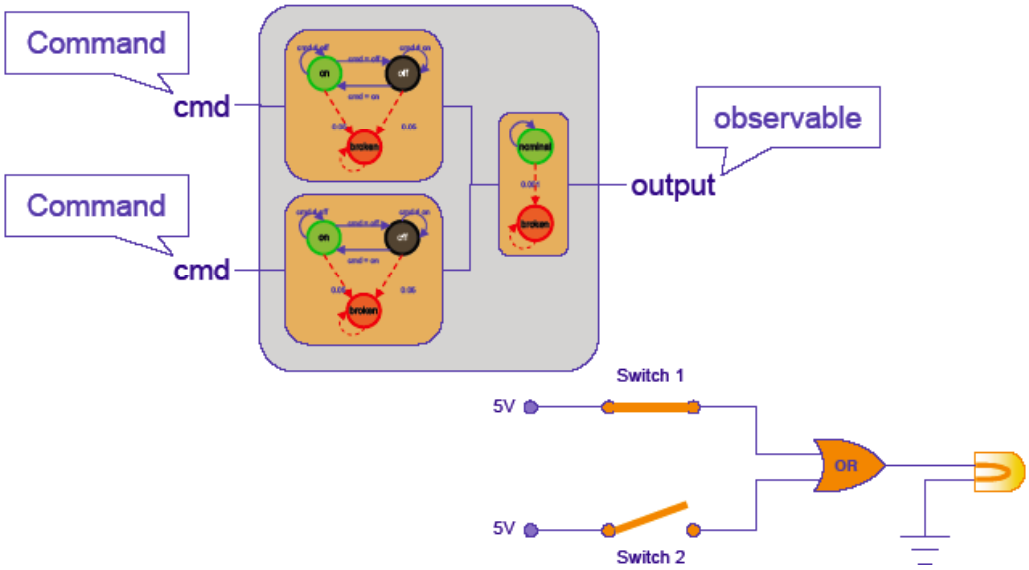


Figure 1. Example of Representation of a Model by a Combination of Automata [Courtesy Seung Chung and Oliver Martin]

Figure 1 illustrates the combination of automata associated with the model for a two-switch simple electrical circuit. Each component of the circuit (Switch1, Switch2 and OR-gate) corresponds to an automaton in the model. The automaton is composed of states and transitions between these states. A state can either be a nominal state or a failure state. The automaton can enter a failure state from any other state by taking a probabilistic transition. Non-probabilistic transitions into nominal states are assigned guards, which are logical propositions depending on the variables of the model. Such a transition can be taken at any moment in time when the assignment to the variables satisfies its guard. Each state is also assigned a *model*, which is a list of assignments to the variables that characterizes that state. The variables in the model can either be *commands* issued to the rover, *observables* output by the rover, or internal dependent variables to which the user does not have direct access.

In order to be usable by Titan, the model is first implemented in LISP, and then fed into a compiler to produce a “MOF-file”, which is a description of the model encoded in MPL (Model Programming Language). MPL is a programming language originally developed at NASA Ames to be used by Livingstone, a model-based diagnosis system designed to monitor and control spacecraft and exploration rovers. Titan then uses this MOF-file for diagnosis and recovery.

Titan has two functions, and hence can be split into two main components: the “Mode Estimation” and “Mode Reconfiguration” components. “Mode Estimation” takes as inputs the values of the observables and commands of the system, and estimates the most probable state configurations of the different components that are consistent with these observations. Assuming the rover is in the most probable configuration outputted by “Mode Estimation”, “Mode Reconfiguration” then suggests a command that should be issued to the rover in order to achieve a given goal state. Each time a command is executed, the “Mode Estimation” re-estimates the state of the system, using the new values of the observables, and hands out the most probable state to “Mode Reconfiguration” which iteratively suggests another command until the goal is reached. This cycle is illustrated in Figure 2.

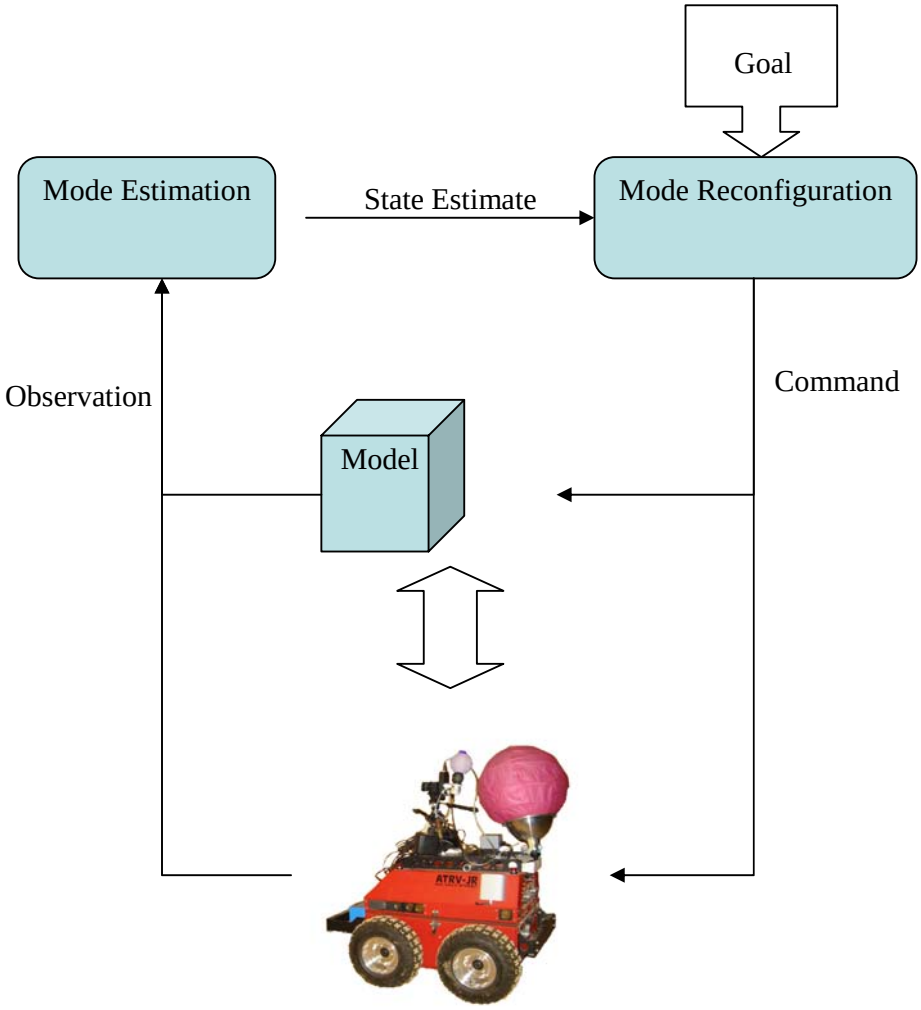


Figure 2. Information Flow Chart through Titan's Components

As illustrated in Figure 2, "Mode Reconfiguration" issues commands that are fed into the model to have it take the transitions from one state configuration to another, during the "propagation step". In theory, the command is also sent in parallel to the rover: in the completely autonomous mode, Titan actually controls the rover by having it execute the commands issued by "Mode Reconfiguration". In a safer semi-autonomous mode, Titan displays the suggested commands and the user manually executes them on the rover. The sensors installed on the rover then output a set of measurements that can be directly observed by Titan through an automated Titan-rover software interface. "Mode Estimation" then checks that these observations fit with the predictions from the model, re-estimating the most probable state of the system if the outputs from the sensors and the model are inconsistent.

2.2 Brief Description of the Rovers

The rovers we worked on are ATRV-Jr rovers, as shown in Figure 3. The rovers have a large variety of sensors that allow the user (and Titan) to monitor the state and the motion of the system. The sensors we actually considered are the laser scanner and the sonar sensors to determine the velocity of the rover, and the wheel encoders to measure the rotation of the wheels. We also used a battery sensor, which is not mentioned on the figure because it observes the internal state of the system, whereas the previously mentioned sensors measure the state of the environment around the rover. Note that the rover has a differential drive, but it was decided only to consider one wheel in our model. This is based on the assumption that all four wheels move together, thus limiting the model of the rover to straight line motion.

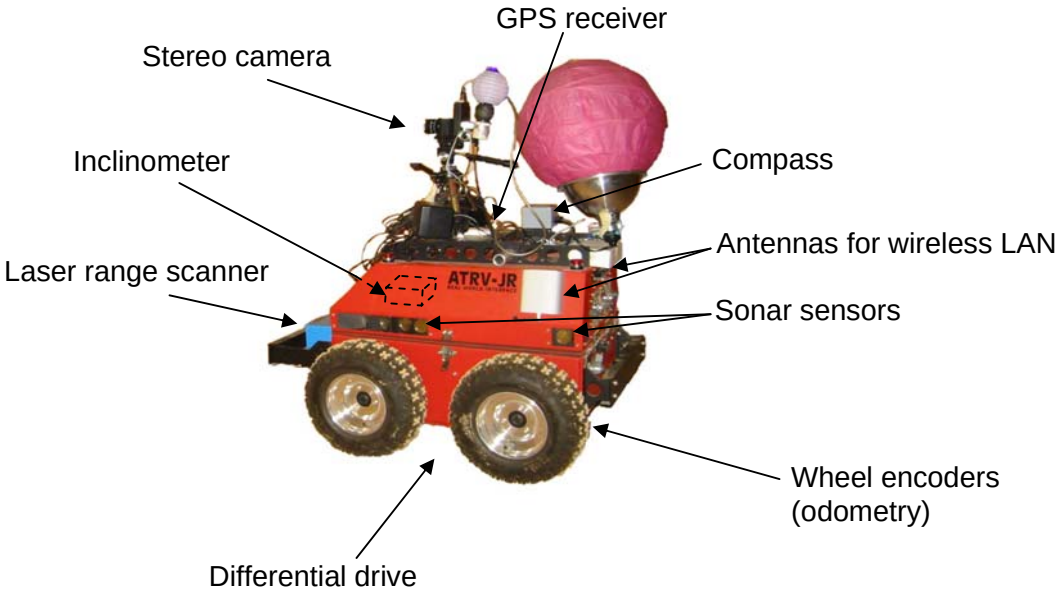


Figure 3. ATRV-Jr Sensors and Actuators
[Courtesy Seung Chung]

The rover is controlled by an onboard computer that sends commands to the actuators via the rFlex board. The rFlex board is an embedded controller at the interface between the motor and the PC. It also gathers information form various sensors and sends it to the PC. Some sensors, however, are connected directly to the PC. The overall internal structure of the rover is presented in Figure 4.

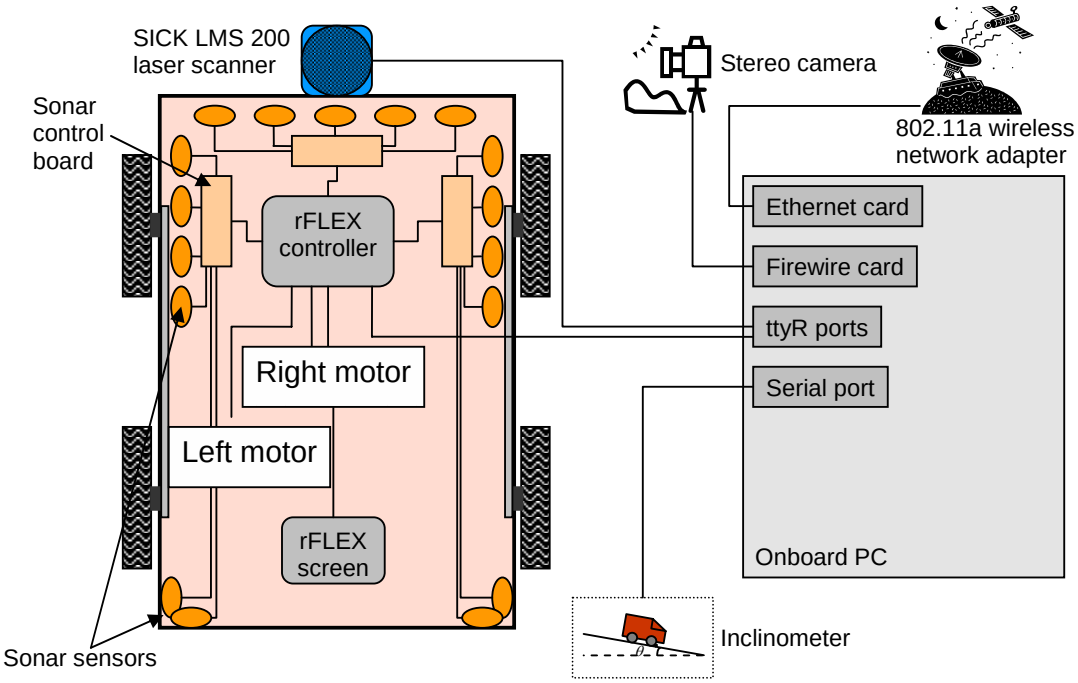


Figure 4. Overall Internal Structure of the Rovers
[Courtesy Seung Chung]

In our consecutive modeling approaches, we tried to come up with models that captured the behavior of components relevant to the scenario in which we wanted Titan to be able to estimate and reconfigure. The description of this baseline scenario is provided in the following section.

3 The Slipping/Sliding Diagnosis Scenario

The environment-dependent failure that needed to be modelled was the case when the rover's motion relative to the environment is inconsistent with the motion of the wheels, either because the rover is slipping or sliding. This situation is described in the following figure:

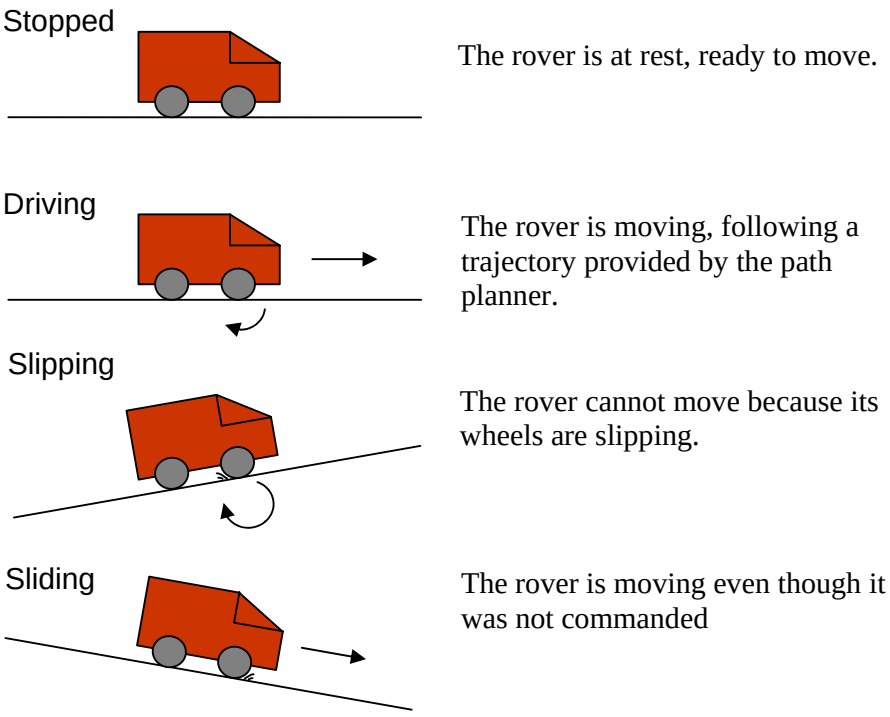


Figure 5. Four Motion Modes to be Diagnosed

Figure 5 represents the four motion modes the onboard PC should be able to discriminate. The first mode is the "stopped" mode, when the wheels are not rotating and the rover is at rest. The second mode is the "driving" mode, in which the wheels are rotating and the rover is moving. Those two modes are nominal modes, corresponding to consistent wheel and rover motions.

The two following modes are non-nominal modes, since the wheel motion is inconsistent with the rover motion. The "Slipping" mode represents the configuration in which the wheels are rotating, but the rover is still because the wheels are slipping. The similar "Sliding" mode represents the configuration when the wheels are not rotating but the rover is moving, presumably because of the slope, and the rover is sliding.

In order to be able to discriminate between those four modes, the onboard PC requires data from two independent sensors. The first sensor is the odometer, measuring the rotation speed of the wheels. A second sensor is required to sense whether the rover is moving or not. There are several sensors available on the rover that can be used in this purpose. The sonar sensors are designed to allow the PC to "see" obstacles or objects around the rover and to evaluate their shape and distance from the vehicle. Variations in these distance measurements can be interpreted as indicators of motion of the rover (assuming that the rover evolves in a motionless

environment, and especially that no other moving rover is in view of the sonar sensors). The laser scanner installed at the front of the truck can also be used in a similar way. This sensor measures the distance from the rover to the closest object in varying directions within the range of the scanner. If the onboard PC observes variations in the distance to the closest object in a given direction, it can infer from these observations that the rover is moving. Eventually, an evaluation of the rover velocity can be made using a more complex combination of hardware and software components. SLAM is a software component that derives a map of the environment and the position of the rover within this environment from the reading on various sensors. In particular, it uses odometry to update the position of the rover, so it is not independent from the motion of the wheels, and this method might not be appropriate to evaluate the velocity of the rover in case of slipping or sliding failures.

As a consequence, the first basic model of the rover included two sensor components: the “sonar” and the “encoder” (standing for the odometer). The third component is the “wheel”. The aim of the mode evaluation problem is for Titan to infer the state of this third component from the outputs of the two sensors. The aim of the mode reconfiguration problem is to send appropriate actions to this third “wheel” component in order to have it transition from a failure state back to a nominal state. Figure 6 illustrates this basic first model.

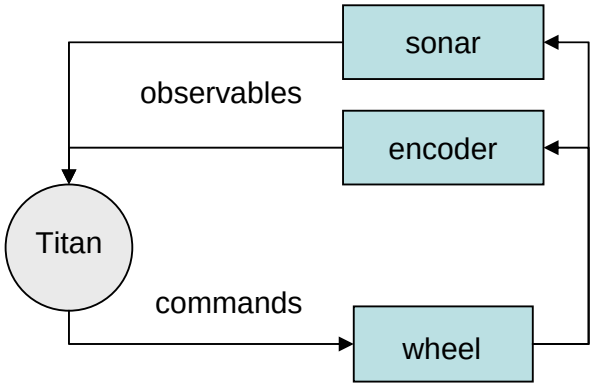


Figure 6. Flow Diagram for the Output -> Input Approach

This model scheme was the starting point of our work. As progress was made in modeling the system, several other components were added in order to account for the complexity of the rover. But several consecutive drastic changes were also made to implement the model in MPL in order to overcome fundamental modeling constraints imposed by Titan. These different approaches to the problem are presented and discussed in the following sections.

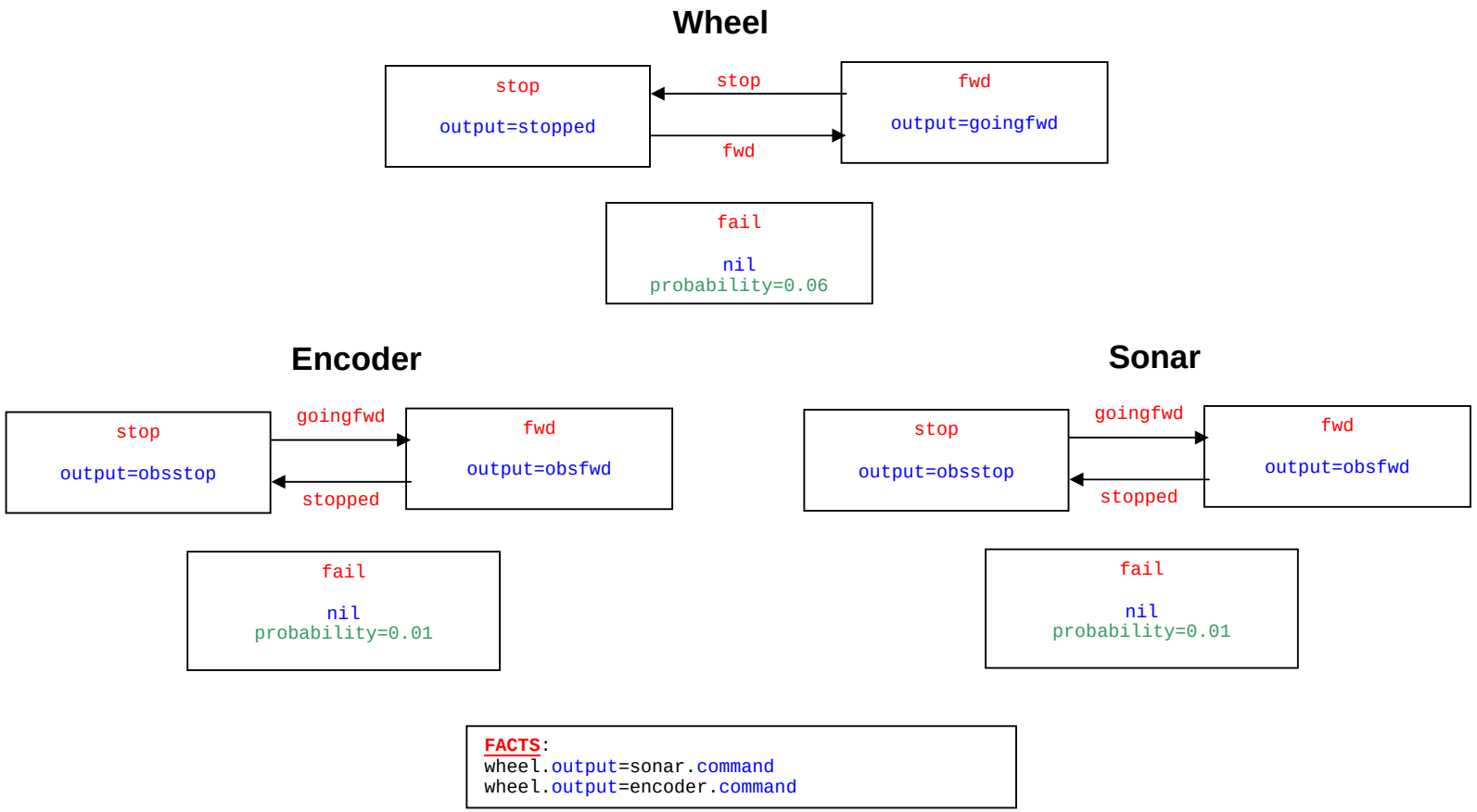
4 First Design Approach: Output -> Input Approach

4.1 Description of the Model

This first method used to implement the basic model illustrated in Figure 6. As suggested by this figure and by the name of the approach, each component was first considered to be a “black box” whose “output” was the “input” of the following component. In this MPL approach, the “outputs” of a component are the variables in the state models, and the “inputs” are the guards on the transitions between these states. They are linked one to the other using relationships listed in the model facts. In the case of the rover the output of the “wheel” component is made equal to the guards on the transitions of the two sensor components, as illustrated in Figure 7. The MOF file for this model can also be found on the enclosed CD and is entitled “version A”.

Note that this model is a very simple model; it has no notion of “slipping” nor “sliding” and cannot be used to diagnose those two failures. But it is the first step we took, and it allows us to underline a major flaw in the design approach, as discussed in the “Critique” section. The “persist” transitions are not shown for clarity.

Figure 7. Output -> Input High Level Model Diagram



In the approach described in Figure 7, the MPL model has one single input, which is put as a guard on the transitions between the nominal states of the "wheel" component. A command "stop" makes the wheel transition to the "stop" state, and a "fwd" command makes it transition to the "fwd" state. The third state is a "fail-mode" state, with a given probability of occurrence (it describes unexpected hardware wheel failures). Each of the ok-mode states has a model linked to the guards on the transitions of the two sensor components. For instance, the guard on the transition from the "stop" to the "fwd" state of the "encoder" component is satisfied if and only if the wheel is in the "fwd" state (or on the "fail" state, which has no model).

The two observables are *encoder.output* and *sonar.output*. Giving the values of these two variables to Titan enables it to estimate the state of the system. For example, if the two observables have the same value "*obs-stop*", Titan will estimate the most probable state of the system, in which the two sensor components are in their respective "stop" states. Then Titan will diagnose that the most probable state for the wheel that is consistent with the two states of the two sensors is the "stop" state, and it infers that the rover is at rest. If on the contrary Titan had been given two different observations from the two sensors, it would have inferred that the wheel was in the "fail" mode, because it has no model, so it is the only state of the wheel consistent with the observations.

4.2 Critique of the Approach

In fact, Titan is not able to estimate the state of this model: if we give Titan the command "*fwd*" and the observations "*sonar.output = obs-fwd*" and "*encoder.output = obs-fwd*", Titan estimates that both sensors have transitioned to their "fail" state.

This error is due to a conceptual flaw in the model, as underlined in Figure 8.

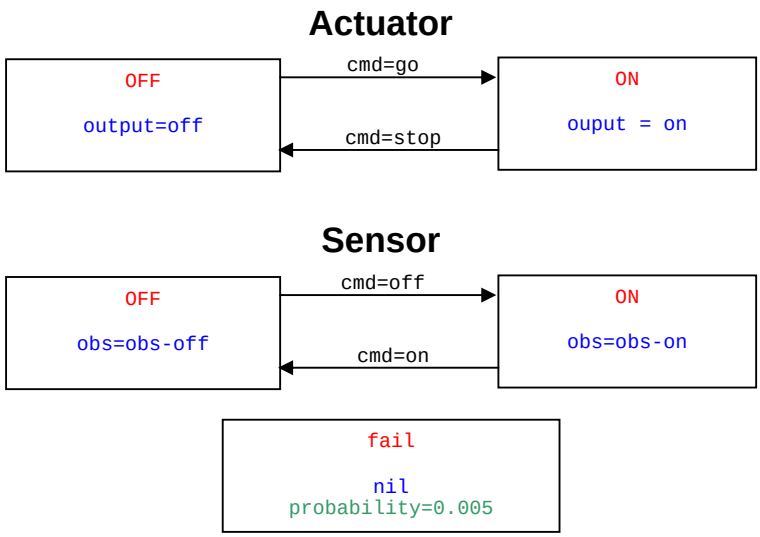


Figure 8. Trivial Output -> Input Two-Component Model

Consider the canonical sensor/actuator model on Figure 8, where the initial configuration of the system is "*off - off*". If Titan is given the command

"*Actuator.cmd=go*" and the observation "*Sensor.obs=obs-on*", then asked to estimate the state of the system, it will first infer from the command that the actuator has transitioned into its "*ON*" state. But Titan will not infer from this that the sensor has also transitioned into its "*ON*" state, because the guard on this transition is not met in the current system state. Since the observable is inconsistent with the model of the "*OFF*" state of the component, and no guarded transition is possible, Titan infers that the component is in its "fail" state, i.e. broken.

This modeling approach is therefore fundamentally flawed. Because of this, another modeling method was considered, the "Output <-> Output" approach, in which the guards on transitions are no longer linked to the output of other component This second approach is described in next section.

5 Second Design Approach: Output <-> Output Approach

5.1 Description of the Model

This approach was chosen to solve the problem of mode estimation described in the previous section. In this model the only transition guards allowed are the guards corresponding to external commands given to Titan. Instead of linking components by creating a relationship between the "output" of one and the "input" of the other (as in the previous "Output -> Input" approach), we chose an "Output <-> Output" approach in which components are linked by their outputs only, and no guard is put on the transitions.

In addition a "motor" component was added in order to increase the detail of the model. This allows successful diagnosis of a greater range of failures.

Figure 9 illustrates the general idea of the second approach.

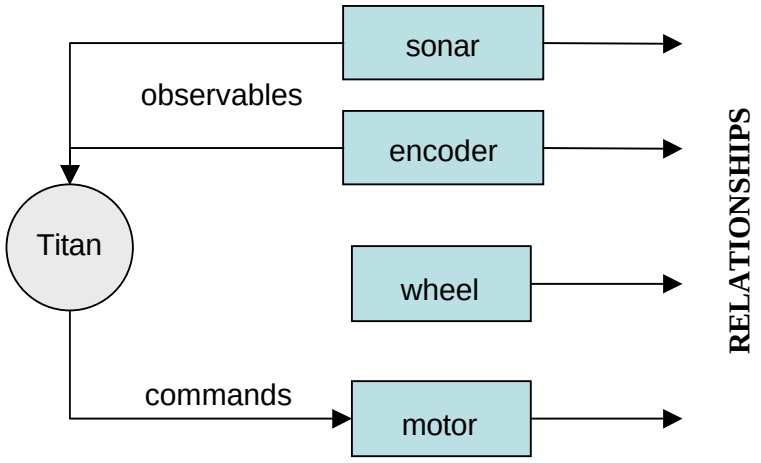


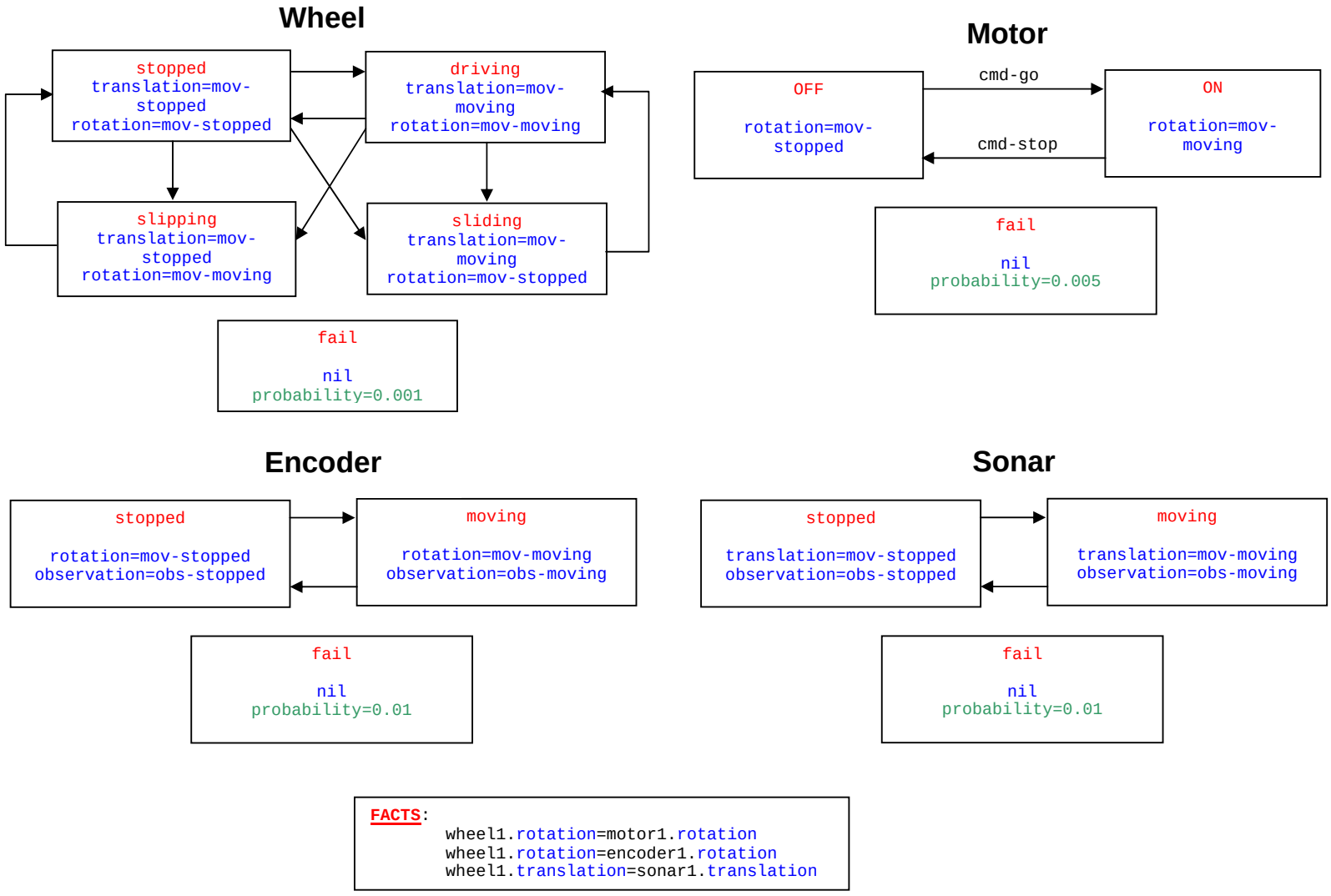
Figure 9. Flow Diagram for the Output <–> Output Approach

This approach solves the estimation issue presented in the previous section. Given the values for the observables, Titan is able to estimate the state of the two sensors. The "outputs" of these two sensors are linked to outputs from the two other components, so that Titan can infer the state of the wheel and the motor directly from

the state of the sensors, without any extra propagation step as required in the first approach. Transition guards appear only in the motor component, and allow the user to command the motor to go from one nominal mode to another. If Titan's first estimation of the state of the system is inconsistent with the previous state and the command given to the motor, Titan infers that at least one of the components has failed and estimates the most probable failure.

A more detailed diagram for this second model is presented on Figure 10 and the corresponding MOF file is the version B of our model and can be found on the enclosed CD.

Figure 10. Output <-> Output High Level Model Diagram



This approach solves all the problems encountered in the previous approach as far as mode estimation is concerned. Mode Estimation is able to go through the following scenario and its ability to estimate the corresponding configurations is demonstrated.

a) Initial state: the rover is at rest

b) Command “go” => successful driving

c) Command “stop” => successful stopping

d) Command “go” => slips

e) Command “stop” => successful stopping

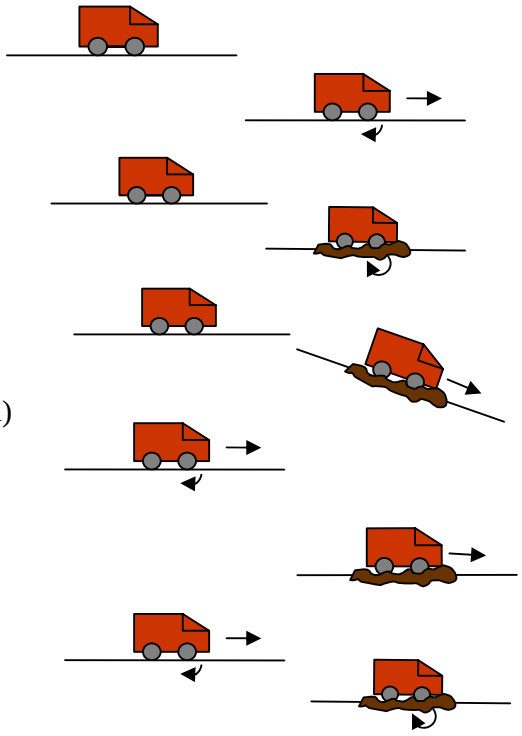
f) Spontaneously starts sliding (no command)

g) Command “go” => successful driving

h) Command “stop” => slides

i) Command “go” => successful driving

j) Spontaneous loss of traction



Mode Estimation also handles the cases when the wheels appear to stop or start rotating spontaneously without being commanded to; Mode Estimation then diagnoses a hardware failure (either a motor failure or a sensor failure, depending on the probabilities).

5.2 Critique of the Approach

The model described here does not model certain transitions between wheel modes which are in fact possible in the real rover. These include the possibility of a slipping wheel spontaneously gaining traction and moving into a driving mode without a command being issued. The second, analogous scenario, is that of a sliding wheel spontaneously stopping without a command being issued.

It seemed initially that this was a simple case of the model not being detailed enough, and that unguarded, probabilistic transitions from slipping to driving and from sliding to stopped could be added It was later determined however that this is fundamentally impossible within the constraints of Titan’s current capabilities. This is discussed in detail in section 9.

Although Mode Estimation is able to carry out correct diagnosis given the commands and observables, Mode Reconfiguration is unable to suggest appropriate action in order to recover from a “slipping” or a “sliding” mode. This is due to a conceptual mistake in the modeling approach. Mode Reconfiguration is unable to handle unguarded transitions. This problem is discussed below.

Consider the initial configuration when the rover is slipping: the wheel is in its “stopped” mode, the motor is “OFF”, the sonar is “stopped” and the encoder is “stopped”. We give Mode Reconfiguration the trivial goal “the wheel component is in its ‘driving’ mode”, that is we want to have the wheels start driving. The transition in the wheel component between the “slipping” and the “stopped” modes is unguarded. Mode Reconfiguration reasons that no command need be issued for this transition to take place. For the motor component to make the transition between its OFF and ON state as required, the command ‘GO’ must be issued. However Mode Reconfiguration maintains a concept of an ordering of the system components. Therefore if the wheel appears first in this ordering, Mode Reconfiguration concludes that it is unable to force the change in the system state and reports that no command is required.

The ordering of system components is visible in this extract from the MOF file created by the compiler. It contains the following lines in the “Variable Definition” section:

```
INSTANCE ENCODER1
    VARIABLE ENCODER1.MODE OF-TYPE STATE WITH-RANGE ENCODER-MODE WITH-NUM 3
[…]
  INSTANCE MOTOR1
    VARIABLE MOTOR1.MODE OF-TYPE STATE WITH-RANGE MOTOR-MODE WITH-NUM 4
[…]
  INSTANCE WHEEL1
    VARIABLE WHEEL1.MODE OF-TYPE STATE WITH-RANGE WHEEL-MODE WITH-NUM 1
[…]
  INSTANCE SONAR1
    VARIABLE SONAR1.MODE OF-TYPE STATE WITH-RANGE SONAR-MODE WITH-NUM 2
```

Manual editing of these numbers may avoid this problem (and proved to solve the problem in this case), but to do so is to do Titan’s job, and moreover in more complex models it might be difficult to see what the correct assignment is. Titan would have put the components in the correct order if they were not all at the same level, i.e. the outputs of some components were inputs to others. But this Output -> Input approach does not handle mode estimation, as was pointed out in the corresponding section.

To solve this issue, we considered a new design approach in order not to use unguarded transitions. This model is described in the following section.

As an aside, the compilation phase also misses all unguarded transitions to nominal modes; these need to be added by hand to the MOF file.

6 Third Design Approach: Distributed Command Output <-> Output Approach

6.1 Description of the model

The two previous approaches were “high level” approaches because they modeled the states of the various components in a “functional” way. For instance, the encoder

component had two separate states to describe its nominal behavior in which it was either outputting “the wheel is rotating” or “the wheel is not rotating”. But the transitions between those two nominal states were unguarded and caused Mode Reconfiguration to fail. In this third modeling approach, those two nominal states have become one single nominal state with disjunctive models. This point of view is more realistic and faithful to the hardware, so that it can be considered a more “low level” approach.

However, all nominal states cannot be melted into one, because we would then have no control over the transitions between these states (in fact, there would be no transitions to nominal states). For instance, the “wheel” component requires separate “stopped” and “driving” states, because the aim of the problem is to specify these states as goal states. So, in order to stick to the idea of a single nominal state, the “sliding” and “slipping” states are now considered as actual fail-mode states, with probabilistic transitions into them. In order to recover from these states, we need transitions to the nominal state; these transitions must be guarded (to avoid reconfiguration problems described in the previous approach), and the guard must not depend on “outputs” from other components (to avoid estimation problems described in the first approach). A solution is to use the external command as a guard on these transitions, which is the reason why this approach is called a “distributed command” approach.

In order to be more faithful to the hardware and to demonstrate our ability to deal with more complex models, we also decided to model the rFlex board. The rFlex board has two main functions: it collects data from the sensors and sends them to the onboard PC, and it also takes commands from the PC and transfers them to the appropriate actuators. This is the reason why the rFlex board was split into two different components, one “rFlexGatherer” gathering data from the sensors, and one “rFlexCommander” receiving commands from Titan. Eventually, in order to limit the complexity of the model, we decided to split the “rFlexGatherer” component further into two components (one for each sensor). The general flow diagram and the detailed transition diagram of this model are represented respectively in Figure 11 and Figure 12. The MOF file for this model is also available on the supplied CD; it is called “version C”.

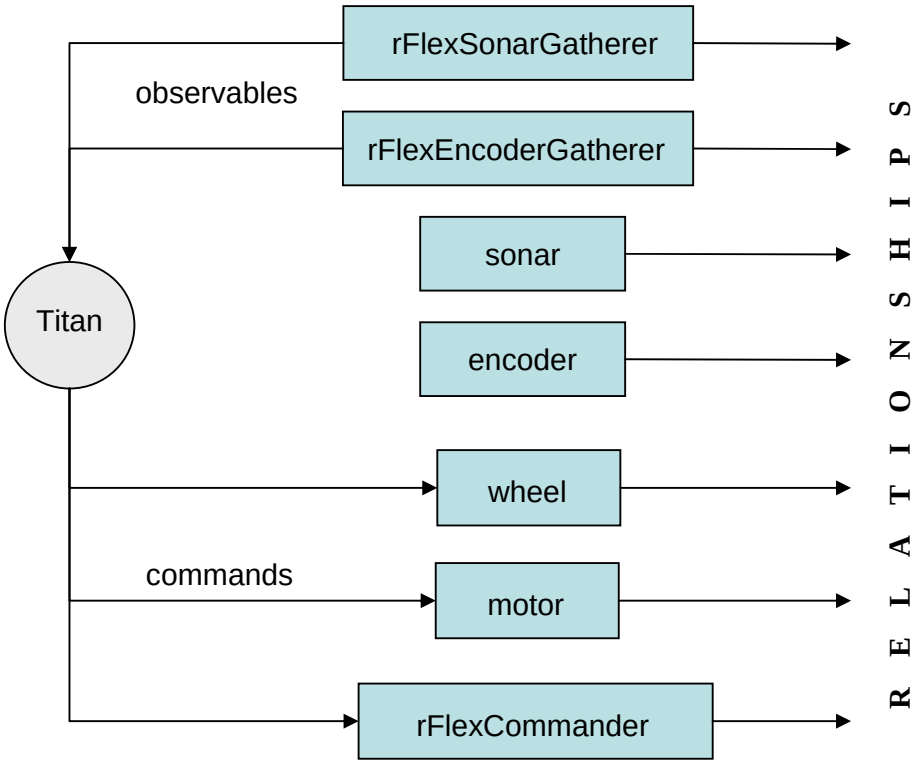


Figure 11. Flow Diagram for the Distributed Command Approach

Figure 12a. Distributed Command Model Diagram

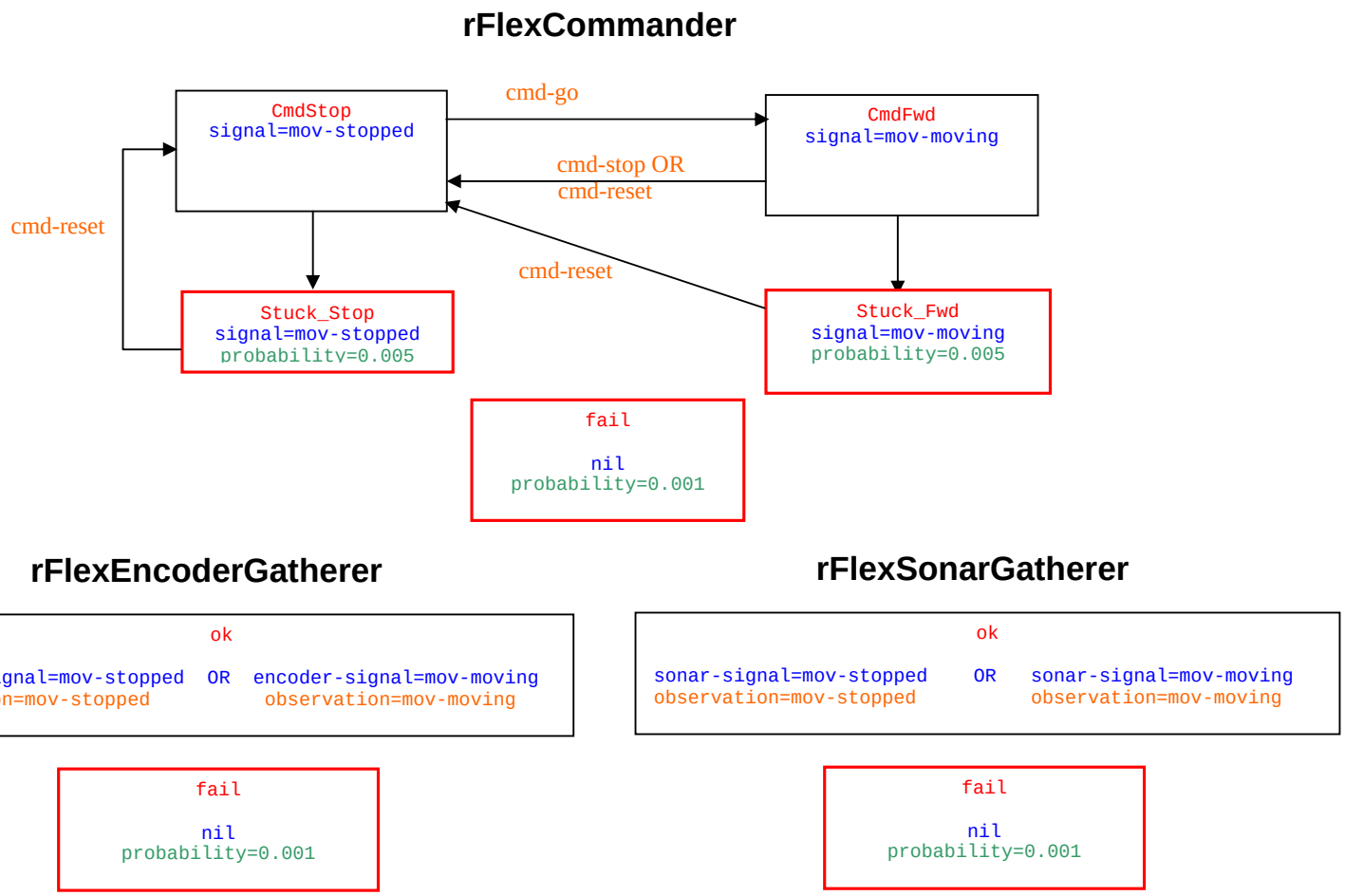
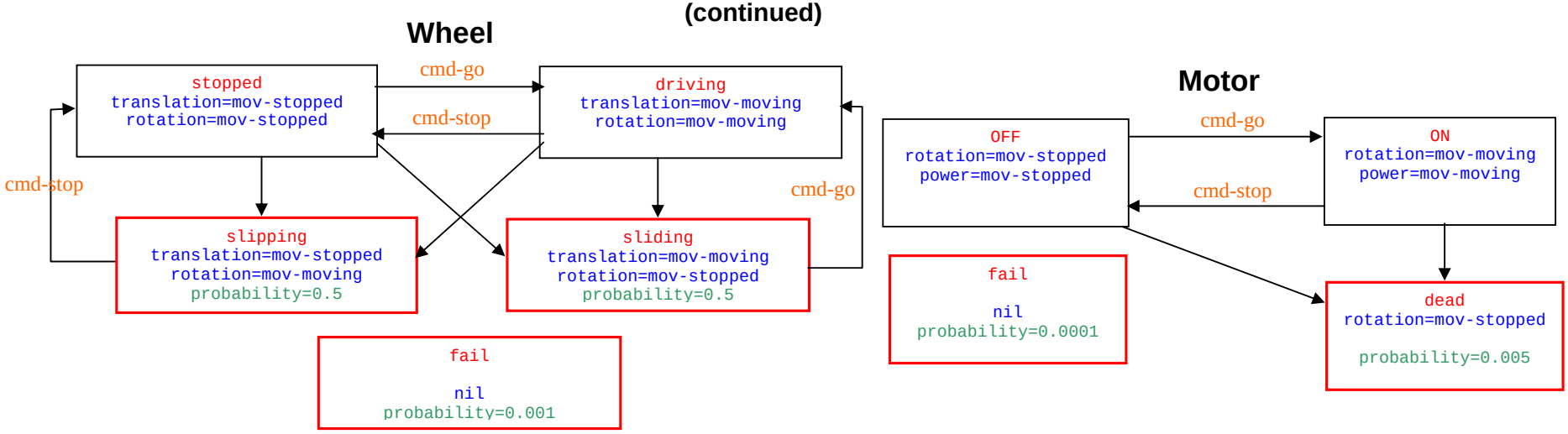
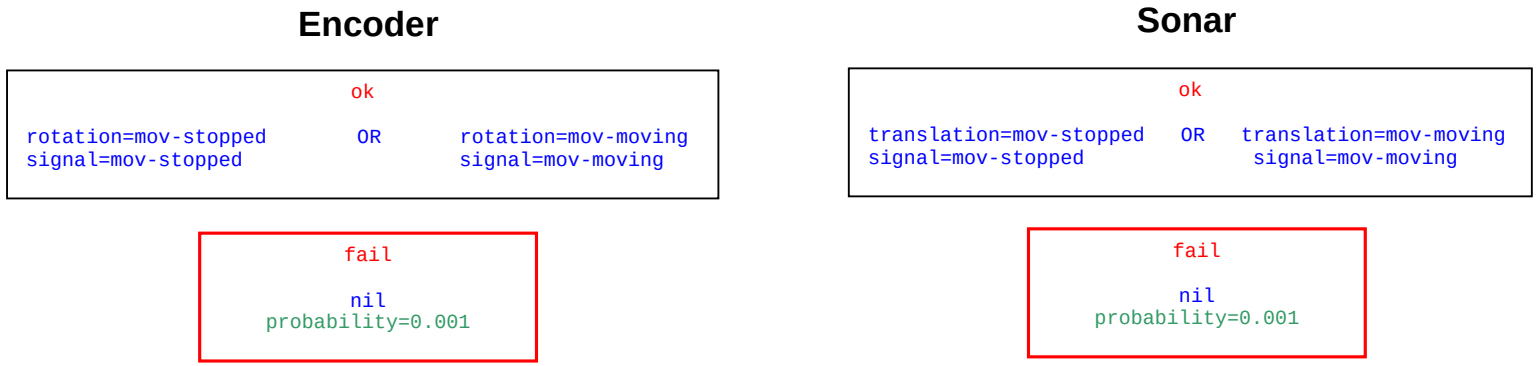
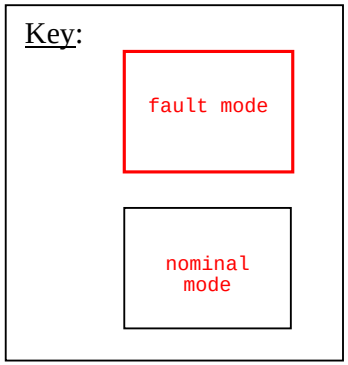


Figure 12b. Distributed Command Model Diagram (continued)







NB:
Within a component there are transitions to the `fail` state from all other states, but these are not shown for clarity. Also all states have `persist` transitions (not shown).

FACTS:

```
wheel1.rotation      =    motor1.rotation
wheel1.rotation      =    encoder1.rotation
wheel1.translation   =    sonar1.translation
sonar1.signal        =    rFlexSonarGatherer1.sonar-signal
encoder1.signal      =    rFlexEncoderGatherer1.encoder-signal
motor1.power         =    rFlexCommander1.signal
motor1.command       =    rFlexCommander1.command
wheel1.command       =    motor1.command
```

As a general rule, the low-level single-OK-mode approach has been enforced for the "passive" components, i.e. the "sensors" ("rFlexGatherers", "encoder' and "sonar"). For the "active" components ("rFlexCommander", "wheel" and "motor") the functional high-level approach prevails, with distributed commands as guards on the transitions. The "rFlexCommander" component has two very special fail-mode states modeling hardware failures when the rFlex board is not responding, and the commands "cmd-go" and "cmd-stop" fail to change the state of the system. It takes a new "cmd-reset" command to recover from those failure states. Those two failure states are very special, because they are probabilistic states, but we do not want them to be reachable from any other state: the "Stuck_Stop" state should only be reachable from the "CmdStop" state. But a probabilistic fail state is by definition reachable from any other state with the same probability. So the MOF file output by the compiler was edited and all undesired transitions to those two failure states were erased by hand.

We have also added a failure state to the "motor" component: the "dead" state is an unrecoverable failure mode modeling the case when the motor is broken and cannot rotate anymore.

6.2 Critique of the Approach

There are still a few problems with this design approach. One of them is that it was assumed that the onboard PC could send a "soft" reset command to the rFlex board, and that this command would only affect the "rFlexCommander" component. This is not realistic, and we fixed this in our fourth approach of the problem. It seems at first glance that we would only need to add the "cmd-reset" command to the guards of the "actuators" to make them disjunctive guards. This was not carried out on this model, because there turned out to be a major problem with the disjunctive reset guard on the "rFlexCommander" component.

Titan simply does not handle disjunctive guards. Any transition with a disjunctive guard in the lisp file is split by the compiler into two separate transitions in the MOF file. Then Titan only considers one of the two transitions, and ignores the second one. The design approach described in the following section deals with this issue by creating new states in order to avoid the use of disjunctive-guard transitions.

However, this model works perfectly well if we ignore the possibility to reset the rFlex board. It is able to estimate correctly the state of each component as we go through the test scenario described in section 4.1. A few reconfiguration tests were made and Titan was able to suggest the appropriate action to recover from the considered failures. More complete and systematic reconfiguration tests were made on the last version of the model, which is described in the following section.

7 Fourth and Latest Design Approach: The Two-Step-Reset Approach

7.1 Brief Description of the Model

This model approach was created in order to be able to reset the rFlex board while avoiding the use of disjunctive transitions that create problems presented in the previous section. The issue was that we were not allowed to have two transitions with different guards ("cmd-stop" and "cmd-reset") between the two same states ("CmdFwd" and "CmdStop"). To solve this problem, we created an intermediate state

through which the “rFlexCommander” component must transition before it can return to a nominal mode. This is equivalent to decomposing the reset procedure into a two-step process. The first step is to actually send the “cmd-reset” command to have the “rFlexCommander” component go into an intermediate state called “initializing” state. Then, as a second step to the reset procedure, the “rFlexCommander” can be sent a command (“cmd-go” or “cmd-stop”) to have it go back into a nominal state of our choice (respectively “go” or “stop”).

Another issue was the fact that the motor and the wheel should be “frozen” in their present state as soon as the rFlex board gets “stuck” and stops responding to commands sent by the onboard PC. This problem is solved by adding a new model variable to the “rFlexCommander” component. This variable is then used as a conjunctive guard in the “motor” and the “wheel’ to make sure that no nominal transition can be taken as long as the “rFlexCommander” is offline, which only happens when it is in one of its two “stuck” states.

The flow diagram corresponding to this latest module is presented on Figure 13.

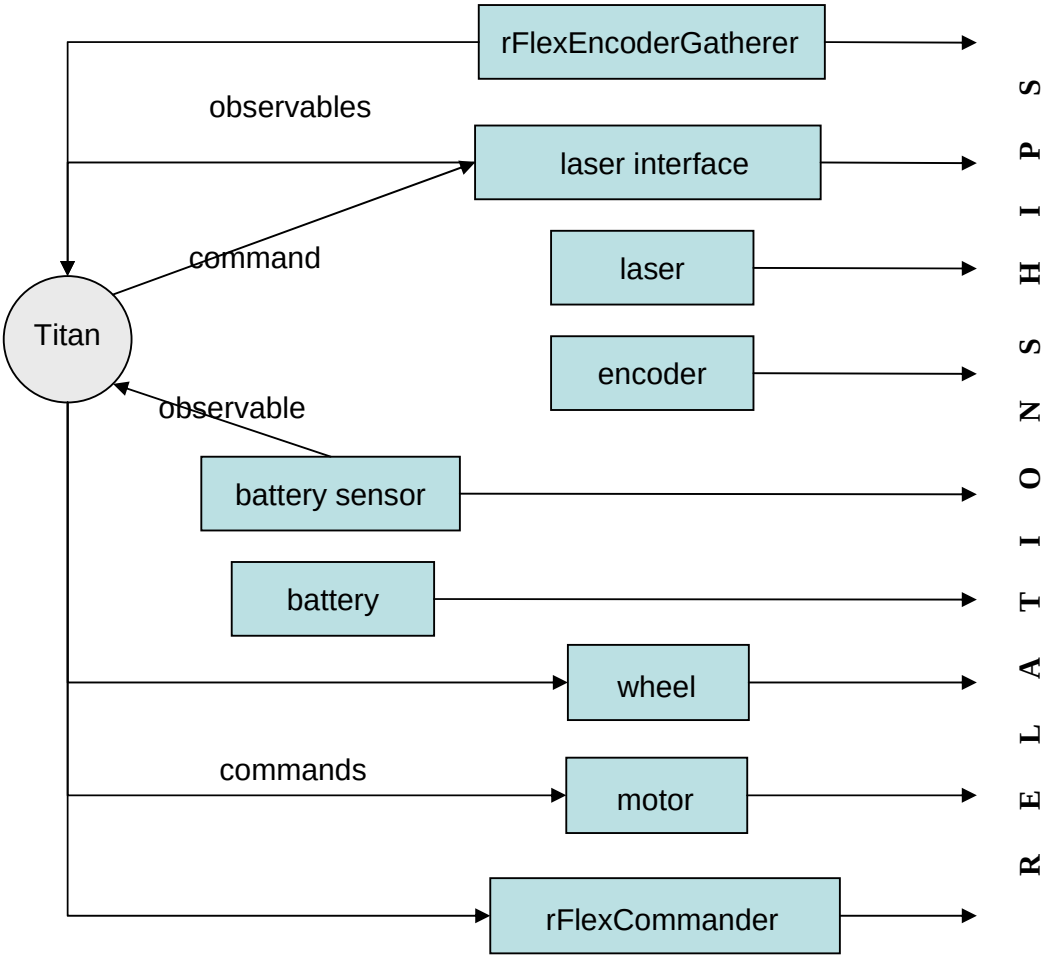


Figure 13. Flow Diagram for the Two-Step-Reset Approach

As it can be seen comparing Figure 11 to Figure 13, there has also been a couple of changes in the components since the previous version of the model. The first change is that the “sonar” and “rFlexSonarGatherer” components have been replaced

by a “laser” component and a “laser interface” component. The decision to make this change was motivated by the suggestion by the TAs that it might be easier to derive the translation motion measurement from the laser than from the sonar. Furthermore, this allowed us to introduce one more recovery possibility, since the laser is independent of the rFlex board and can be sent a reset command via the “laser interface” component, which stands for the ttyR link between the laser and the onboard PC. Two other components were also added to the model in order to account for the possibility of battery discharge or failure. The “battery” component is linked to the motor so that the motor can only be rotating if the battery charge level is high, which can be observed by Titan through the output of the “battery sensor” component.

7.2 Extensive Description of the Model

Since this version of the model is the final version we finally came up with, each component will now be described and explained in detail. The diagram of this model is presented in Figure 14. The .lisp and MOF files can be found on the enclosed CD as version D.

Figure 14a. Transition Diagram for the Two-Step-Reset Approach

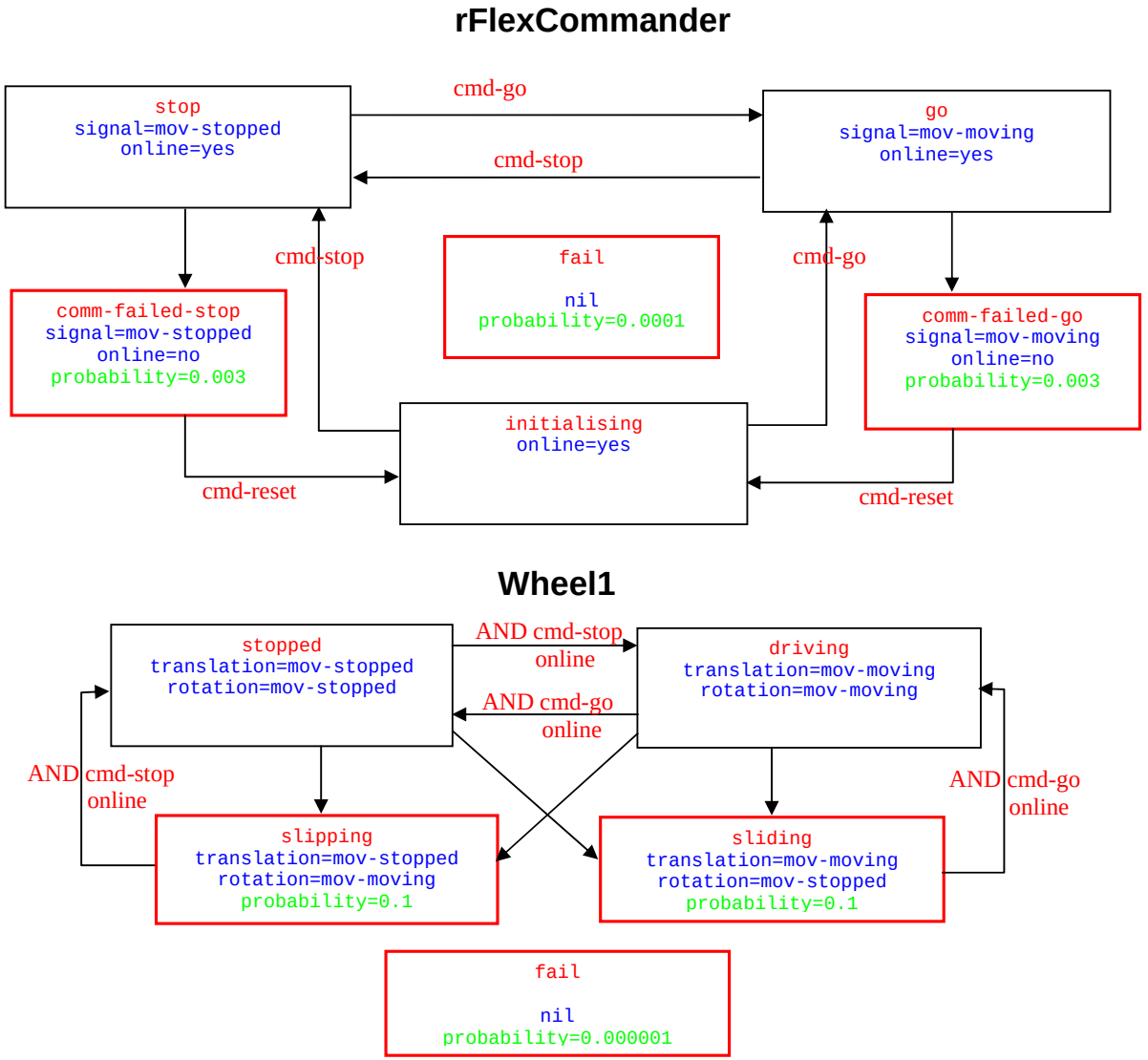


Figure 14b. Transition Diagram for the Two-Step-Reset Approach (continued)

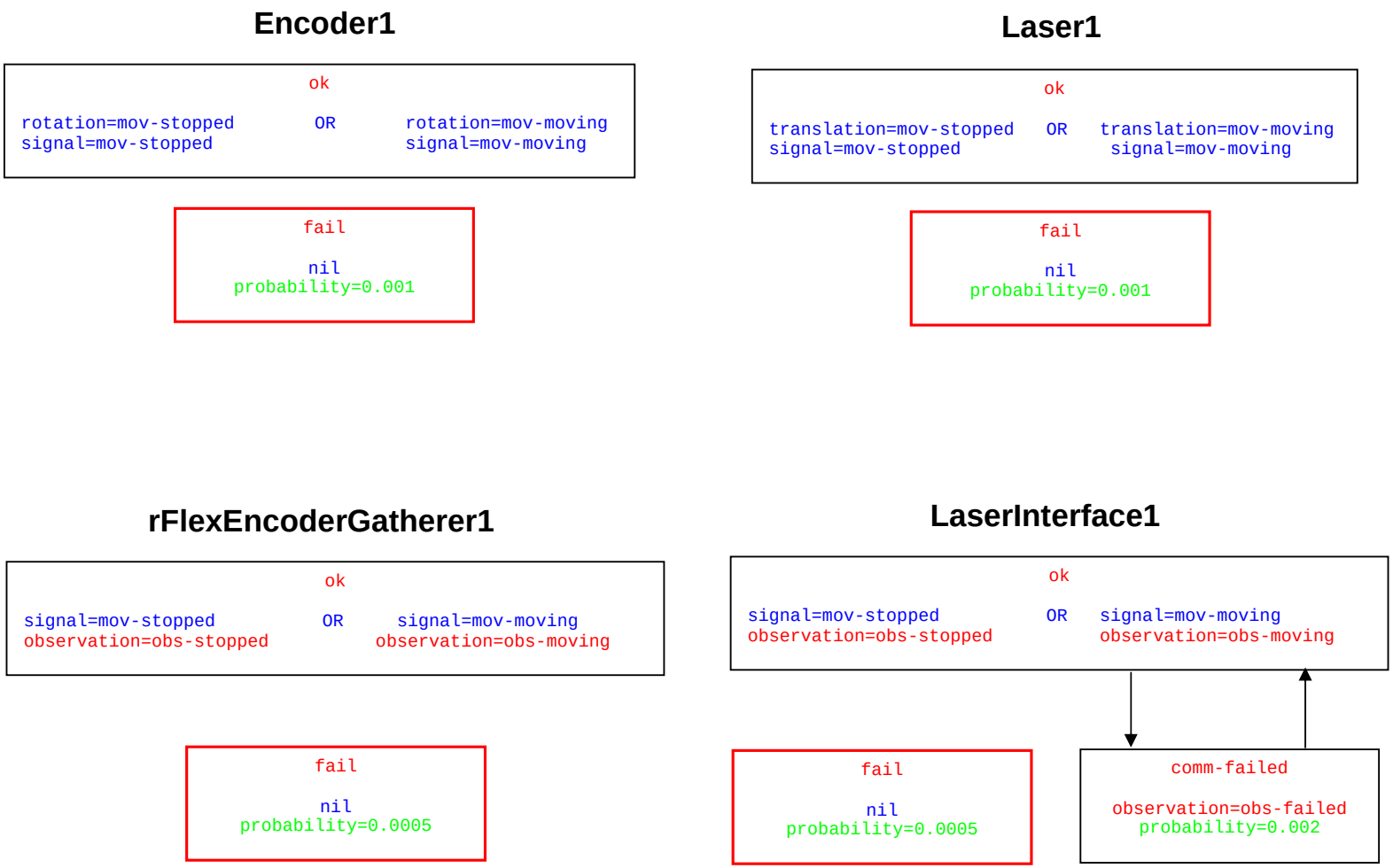
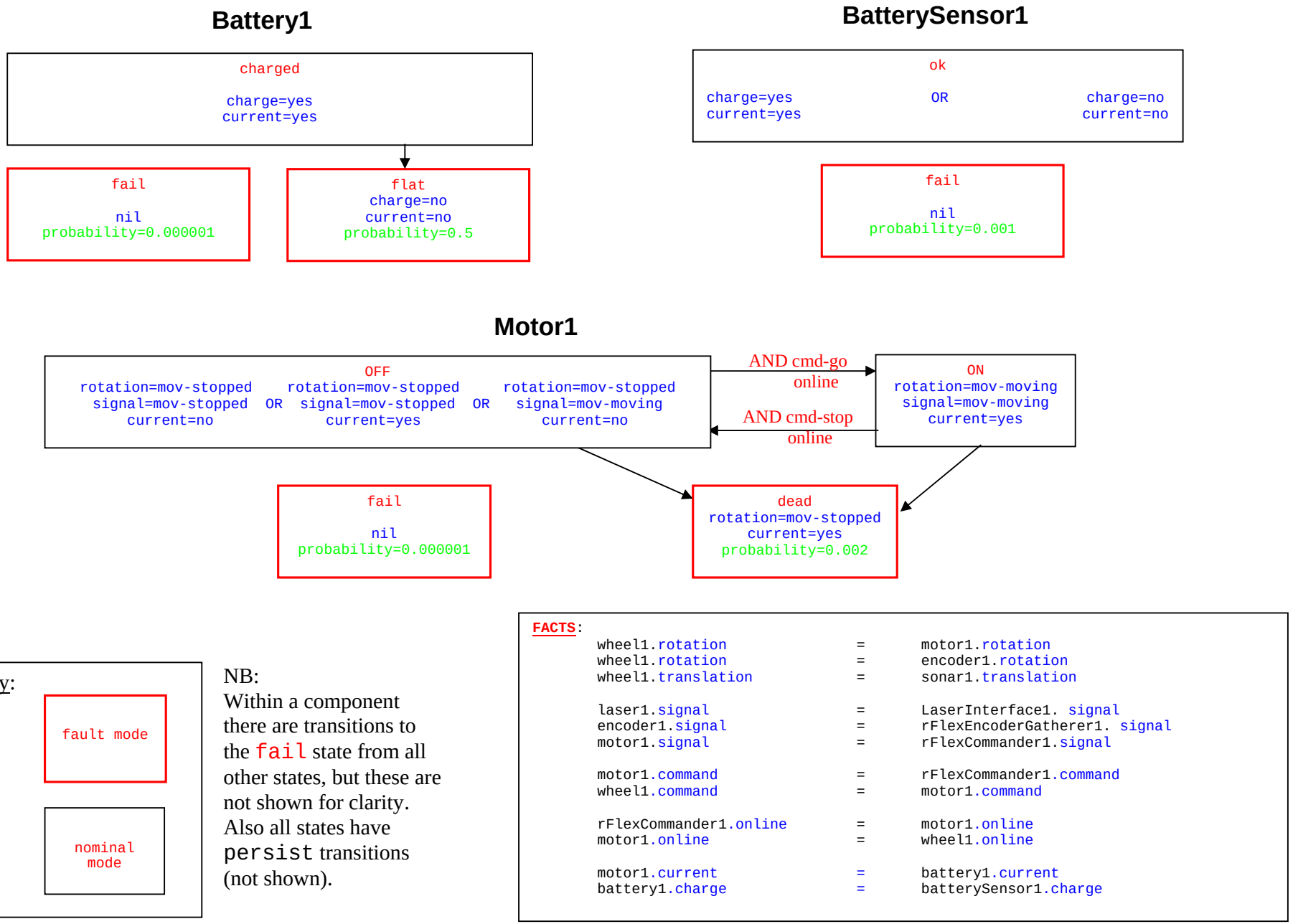


Figure 14c. Transition Diagram for the Two-Step-Reset Approach (continued)



Key:

fault mode

nominal mode

NB:
Within a component there are transitions to the `fail` state from all other states, but these are not shown for clarity. Also all states have `persist` transitions (not shown).

FACTS:

```
wheel1.rotation              =      motor1.rotation
wheel1.rotation              =      encoder1.rotation
wheel1.translation           =      sonar1.translation

laser1.signal                =      LaserInterface1. signal
encoder1.signal              =      rFlexEncoderGatherer1. signal
motor1.signal                =      rFlexCommander1.signal

motor1.command               =      rFlexCommander1.command
wheel1.command               =      motor1.command

rFlexCommander1.online       =      motor1.online
motor1.online                =      wheel1.online

motor1.current               =      battery1.current
battery1.charge              =      batterySensor1.charge
```

7.2.1 rFlexCommander

- "stop" mode: nominal mode modeling the case when the rFlex board is commanding the motor to stay still. The component can take the transition into the "go" mode when it is commanded to go forward.
- "go" mode: nominal mode corresponding to the rFlex board telling the motor to rotate. The rFlexCommander can go back to the "stop" mode through a transition guarded by a "cmd-stop" command issued by Titan.
- "comm-failed-stop" and "comm-failed-go" states: failure states representing special fault-modes. They can be reached with a probability of 0.003, but not from any other mode, as is the default case. "comm-failed-stop" can only be reached from "stop", and "comm-failed-go" from the "go" mode. To impose this, the MOF file has to be edited to remove all undesired probabilistic transitions into these two modes. These modes stand for the case when the rFlex board stops responding, and the system is stuck stopped or stuck going forward, depending on the state of the system before the hardware failure. The two modes impose "online=no", so that all nominal transitions on the wheel and the motor are forbidden, and both components are bound to stay in their present modes (or go into a probabilistic failure mode). We can recover from these two failures by sending a "cmd-reset" command to the rFlexCommander, which then takes the transition into its "initializing" mode.
- "initializing" mode": contrived approach used to avoid disjunctive transitions guards. It also has a physical interpretation: it represents the mode of the rFlex board as it is being reset. The reset procedure is a two-step process consisting of a first "cmd-reset" command, and then a "cmd-go" or "cmd-stop" command. This procedure is described in further details in the next "Critique" section.
- "fail" mode: generic fail mode. Assigned a very low probability designed to handle unexpected behavior or the component.

7.2.2 Wheel

The "wheel" is a component corresponding to the physical wheel of the rover, but it can also be interpreted as representing the overall motion of the Rover, as used in diagnosis and recovery.

- "stopped" and "driving" modes: the two nominal modes of the component. They stand for the two nominal motion of the rover. The transitions between these two modes are guarded by conjunctive clauses involving the command issued by the PC and also the "online" variable. This last variable is used to make sure the two guards are not satisfied when the rFlex board is not responding.
- "slipping" and "sliding" modes: failure modes representing the motion configuration of the rover when it is slipping and sliding respectively. Assuming no hardware failure, these states are diagnosed by Titan when the

observations provided by the two sensors are inconsistent. Their probability is rather high in order to make sure they are a more probable diagnosis than a hardware failure. They are both recoverable: the slipping rover can try to regain traction by taking the transition to the "stopped" mode, and the rover can also start "driving" again to regain traction when it is sliding. Note that no direct reconfiguration from "slipping" to "driving" or from "sliding" to "stopped" is allowed in this model; this point is briefly discussed in the beginning of section 5.2, and further explained in the end of section 9.1.

- "fail" mode: generic unexpected failure mode with very low probability.

7.2.3 Motor

- "ON" mode: nominal mode in which the motor is rotating as commanded by the rFlex board.
- "OFF" mode: nominal mode of the motor standing for the case when the motor is not rotating. The transitions between these two nominal modes are guarded by the command from the PC and also by the "online" variable output by rFlexCommander, to make sure the motor is stuck in its current mode as soon as the rFlex board is no longer responding to the commands issued by the PC.
- "dead" mode: hardware motor failure: the motor is commanded to rotate by the rFlex board, the battery level is sufficient, but the motor is broken and does not rotate. This failure is not recoverable.
- "fail" mode: generic unexpected failure mode.

7.2.4 Encoder, Laser, BatterySensor, rFlexEncoderGatherer and LaserInterface

These four components are very similar because they are all passive sensors, except for LaserInterface, which has a reset capability.

- "ok" mode: this is the single nominal mode in which the sensor behaves correctly, outputting the correct mode of the component or environmental parameter it monitor. The encoder monitors the motion of the wheel while the laser detects the translational motion of the rover. The rFlexEncoderGatherer and LaserInterface components represent the interface between the sensor hardware and the onboard PC. The BatterySensor monitors the charge level of the battery.
- "fail" mode: generic unexpected failure mode, standing for an unconstrained hardware failure.
- "comm-failed" mode: this mode is particular to LaserInterface. It models a failure of the ttyR link between the laser and the PC. This failure is recoverable by a reset command.

7.2.5 Battery

This component is very similar to LaserInterface, except its second failure mode (standing for a “flat” or “dead” battery) is not recoverable. Note that the battery considered here is only the battery supplying power to the motor. We are not modeling the power supply necessary to run the rFlex board and the sensors.

7.3 Critique of the Approach

At first sight, it seems the solution found to the conjunctive guards issue is a step backwards: the “*online*” variable that was created is an “output” of the “rFlexCommander” component, and is used as a guard on transitions in all the other “active” components. So this goes back to the “Output -> Input” design approach, which was proven to be flawed. However, the rFLexCommander does use dependent variables to tie its modes to the modes of the motor, so Mode Estimation will still function correctly. Use of the “online” variable is a poor approach in general, but the component has been carefully constructed to avoid any problems. As a result, this method has proved successful at implementing a reset function without the use of disjunctive transition guards. This capability is demonstrated in the following mini-scenario.

Assume Titan has diagnosed that the “rFlexCommander” component was in its “comm-failed-go” state, i.e. the rFlex board has stopped responding as the rover was going forward. The “*online*” variable is equal to “no”, so that all nominal transitions in the actuators are blocked and all actuators are stuck in their present nominal state corresponding to the rover going forwards. The reset command is then sent to the rFlex board, so that the “rFlexCommander” component enters its “initializing” state. The value of the “*online*” variable then changes to “yes”, so that all nominal transition guards in the actuators become satisfiable. If, at the very same instant, the command sent to the rover is equal to “cmd-stop”, then those guards become satisfied, but the transitions are not taken, because Titan needs another propagation step to do so. But this is fine, because it models the fact that the rFlex board is still unable to pass the commands from the onboard PC to the actuators while it is being re-initialized. In order to complete this re-initialization, the rFlex board is eventually sent a “stop” or “go” command. This command is also distributed to the other actuators, which are then able to take the corresponding transition in one single propagation step.

Another last but important critique of this design approach is that the MOF file output by the compiler still needs editing in order to assign the number 1 to the “rFlexCommander” component, as described in a previous section. This is required to ensure Titan’s ability to suggest the reset command when we try to recover from a rFlex board communication failure. The MOF file also has to be edited in order to delete all undesired probabilistic transitions into the two special stuck states of the “rFlexCommander” component.

8 Diagnosis and Recovery Scenarios

8.1 Introduction

This section describes a number of diagnosis and recovery scenarios which were designed to demonstrate the strengths and weakness of the rover model and the Titan system. Results of tests and conclusions are also presented.

8.2 Scenarios

8.2.1 Slipping and Sliding

In this scenario the rover's components are all functioning correctly. Due to environmental conditions, the rover's wheels lose traction at certain instants. This causes the rover to slip and slide. Titan's task is to diagnose whether the rover is slipping, sliding or driving and suggest the correct sequence of commands to issue in order to recover to the desired goal state.

As described previously, successful diagnosis and recovery in this scenario is the baseline aim of the project. The results of testing are shown in Figure 15.

The results show that in all cases Titan is able to determine the state of the wheel, and hence the rover, correctly. Given a goal state, Titan is able to suggest a sequence of commands to achieve that goal. For example, in the case where the wheel is slipping, Titan determines that first the wheels must be stopped to regain traction, and then the 'go' command should be issued to start driving.

The scenario above could be extended to include various sequences of slipping and sliding. For example on an incline, the rover could start sliding spontaneously. In all cases where the sequences are possible given the transitions in the wheel component model, Titan is successful in both diagnosis and recovery.

There are certain sequences which are not consistent with the wheel model. For example, the wheel automaton cannot make a transition between slipping and driving; this transition would prevent Mode Reconfiguration from working as described in section X. Hence a scenario where the wheel makes this transition without a stop command followed by a go command being issued would cause Mode Estimation to fail. In this case Titan would determine incorrectly that the wheel had entered the unmodelled 'fail' state.

This, however, is a fundamental limitation of Titan as discussed in section 9.

Description	Observations	State Estimated by Titan	Requested Goal	Command from Titan
Initialise with rover stopped, all components in ok mode.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF All components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to move. Rover has traction and moves successfully.	BatterySensor1: Charged LaserInterface1: Moving RFlexEncoderGatherer: Moving	Wheel1 is DRIVING Motor1 is ON All components are OK	Wheel1=STOPPED	Send STOP command to rFlexCommander1
Command rover to stop. Rover stops successfully.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF All components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to move. Rover has insufficient traction and wheels slip.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Moving	Wheel1 is SLIPPING Motor1 is ON All components are OK	Wheel1=DRIVING	Send STOP command to rFlexCommander1
Command rover to stop. Rover stops successfully.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF All components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to move. Rover has regained traction and moves successfully.	BatterySensor1: Charged LaserInterface1: Moving RFlexEncoderGatherer: Moving	Wheel1 is DRIVING Motor1 is ON All components are OK	Wheel1=STOPPED	Send STOP command to rFlexCommander1
Command rover to stop. Rover has insufficient traction and slides.	BatterySensor1: Charged LaserInterface1: Moving RFlexEncoderGatherer: Stopped	Wheel1 is SLIDING Motor1 is OFF All components are OK	Wheel1=STOPPED	Send GO command to rFlexCommander1
Command rover to move. Rover drives and regains traction.	BatterySensor1: Charged LaserInterface1: Moving RFlexEncoderGatherer: Moving	Wheel1 is DRIVING Motor1 is ON All components are OK	Wheel1=STOPPED	Send STOP command to rFlexCommander1
Command rover to stop. Rover has traction and stops.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF All components are OK	Wheel1=STOPPED	No command necessary
No command. Rover stays stationary.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF All components are OK	None	None

Figure 15. Results of Slip and Slide Scenario Testing

8.2.2 Laser Hardware Failure

This scenario involves a rover where the laser component has failed in such a way that it always produces a ‘stopped’ signal regardless of the motion of the rover. This is different from the failure where the laser interface to the PC has a communication failure, and is a more difficult diagnosis problem because the PC cannot detect the failure directly.

In the scenario the rover starts from rest. The goal throughout is that the rover is moving. Titan’s aim is to estimate the state of the rover and determine which commands should be issued in order to achieve this goal. Figure 16 shows the results of testing this scenario.

The results show that at first Titan correctly determines that the ‘go’ command should be issued. Once this occurs, the rover does in fact move. Due to the faulty laser component, however, the observations show that the rover is stationary despite the fact that its wheels are turning.

Mode Estimation decides that the most likely diagnosis is that the wheels are slipping. This is the correct diagnosis given the observations; the probability of the wheels slipping is far greater than the probability of a laser component failure.

Titan determines that the recovery trajectory from slipping to driving, the goal state, is to issue the stop command followed by the go command. When this is done, however, the faulty laser gives a ‘stopped’ signal, and once more Mode Estimation determines that the most likely diagnosis is that the rover is slipping.

This cycle repeats itself a number of times. However during this process, Titan retains a tree of possible trajectories to the current possible states of the system. Eventually Titan deduces that it is more likely that the laser component had been in the ‘failed’ state from the beginning of the test than that the wheels slipped every time the go command was issued.

At this point, Titan determines that the rover is in fact driving. Since this was the original goal state, no command need be issued.

This scenario therefore demonstrates Titan’s capability to track multiple trajectories through the system’s state space and deduce the most likely assignment of modes. Unlike the slip and slide scenario, where the observations and commands uniquely determine the state of the wheel (ignoring the unmodelled ‘fail’ states which have very low probabilities), this scenario requires Titan to use the probabilistic model of the system along with multiple observations over time to make the correct diagnosis. This is a strength of the Titan system.

Description	Observations	State Estimated by Titan	Requested Goal	Command from Titan
Initialise with rover stopped. Laser component has failed so that it always detects that the rover is stopped. All other components ok.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF Laser1 is OK All other components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to drive. Rover drives successfully, but laser component continues to give STOPPED signal.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Moving	Wheel1 is SLIPPING Motor1 is ON Laser1 is OK All other components are OK	Wheel1=DRIVING	Send STOP command to rFlexCommander1
Command rover to stop. Rover stops successfully.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF Laser1 is OK All other components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to drive. Rover drives successfully, but laser component continues to give STOPPED signal.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Moving	Wheel1 is SLIPPING Motor1 is ON Laser1 is OK All other components are OK	Wheel1=DRIVING	Send STOP command to rFlexCommander1
Command rover to stop. Rover stops successfully.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF Laser1 is OK All other components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to drive. Rover drives successfully, but laser component continues to give STOPPED signal.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Moving	Wheel1 is SLIPPING Motor1 is ON Laser1 is OK All other components are OK	Wheel1=DRIVING	Send STOP command to rFlexCommander1
Command rover to stop. Rover stops successfully.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF Laser1 is OK All other components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to drive. Rover drives successfully, but laser component continues to give STOPPED signal.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Moving	Wheel1 is DRIVING Motor1 is ON Laser1 is FAILED All other components are OK	Wheel1=DRIVING	No command necessary

Figure 16: Results of Laser Hardware Failure Scenario Testing

8.2.3 Laser Interface Communication Failure

In this scenario, the laser interface develops a communication fault with the PC. This form of fault can be observed directly by the onboard PC.

Again, the goal throughout is for the rover to be driving. The results of testing are shown in Figure 17.

This is a simple scenario, because the communication failure is directly observable by the onboard PC. Titan makes the correct diagnosis and determines that the reset command should be sent to the laser interface.

Description	Observations	State Estimated by Titan	Requested Goal	Command from Titan
Initialise with rover stopped. All components ok.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF LaserInterface1 is OK All other components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to drive. Rover drives successfully, but a communication failure has occurred between the laser interface and onboard PC.	BatterySensor1: Charged LaserInterface1: Failed RFlexEncoderGatherer: Moving	Wheel1 is DRIVING Motor1 is ON Laserinterface1 is FAILED All other components are OK	Wheel1=DRIVING	Send RESET command to LaserInterface1
Send RESET command to laser interface. Laser interface resets and communication reinitialises successfully	BatterySensor1: Charged LaserInterface1: Moving RFlexEncoderGatherer: Moving	Wheel1 is DRIVING Motor1 is OFF LaserInterface1 is OK All other components are OK	Wheel1=DRIVING	No command necessary

Figure 17. Results of Laser Interface Communication Failure Scenario

8.2.4 rFlexCommander Communication Failure

In this scenario the rFlex board has a communication failure and hence stops responding to commands from the onboard PC.

The rover starts at rest, with all other components functioning correctly. The goal throughout is for the rover to be driving. Titan's task is to diagnose the communication fault and determine the correct recovery trajectory. The results are shown in Figure 18.

After the first 'go' command, the observations show that neither the wheels are turning nor is the rover moving. At this stage there are a number of consistent mode assignments. One set of assignments involves a dead motor. Another involves a dead battery, but also a failed battery sensor, since the battery sensor indicates that the battery has charge.

The most likely assignment, however, involves the rFlexCommander being in the 'comm-failed-stop' mode. This mode models the state where the commander is stuck commanding the motor to stop, which is in fact the case in this scenario.

The results show that Mode Estimation correctly determines the latter case as being the most probable, and Mode Reconfiguration states that the reset command should be sent to the rFlex board. Resetting the rFlex board takes it into the initialising mode. Mode Estimation determines that this has happened, and issues the 'go' command, which is the correct recovery strategy.

Description	Observations	State Estimated by Titan	Requested Goal	Command from Titan
Initialise with rover stopped, all components in ok mode.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF rFlexCommander1 is STOP All other components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to move. rFlexCommander board has a communication failure and does not respond.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF rFlexCommander1 is COMM-FAILED-STOP All other components are OK	Wheel1=DRIVING	Send RESET command to rFlexCommander1
Send RESET command to rFlexCommander board. Board goes into initialising mode.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF rFlexCommander1 is INITIALISING All other components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to move. Rover moves successfully.	BatterySensor1: Charged LaserInterface1: Moving RFlexEncoderGatherer: Moving	Wheel1 is DRIVING Motor1 is ON All components are OK	Wheel1=DRIVING	No command necessary

Figure 18: Results from rFlexCommander Failure Scenario

8.2.5 Dead Motor

In this scenario, the rover has a dead motor, which will not turn the wheel even if supplied with current.

As in the previous scenario, the rover starts from rest, and the goal throughout is to be in the ‘driving’ state. Results are shown in Figure 19.

After the initial ‘go’ command is issued, the observations in this scenario are exactly the same as in the rFlex communication failure scenario. As such, Titan estimates that the most likely cause is a communication failure in the rFlex board, and decides that resetting the board is the best recovery strategy.

After the reset cycle has been carried out, however, the rover still fails to respond to a ‘go’ command. In the rFlexCommander model, the transition between the ‘initialising’ mode and the ‘go’ mode occurs with a 100% probability if the ‘go’ command is issued. Hence for the initial diagnosis of an rFlex communication failure to be correct, the rFlexCommander must be in the ‘go’ mode after the reset. Without some other failure, this is inconsistent with the observations. Hence Titan determines that the most likely diagnosis is that the motor was dead from the beginning.

Given this diagnosis, Titan correctly reports that no recovery is possible since the motor cannot return to a nominal mode.

Description	Observations	State Estimated by Titan	Requested Goal	Command from Titan
Initialise with rover stopped. Motor is dead and will not move even when supplied current. All other components ok.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF rFlexCommander1 is STOP All components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to move. Due to dead motor, rover remains stationary.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF rFlexCommander1 is COMM-FAILED-STOP All other components are OK	Wheel1=DRIVING	Send RESET command to rFlexCommander1
Send RESET command to rFlexCommander board. The board resets.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF rFlexCommander1 is INITIALISING All other components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to move. Rover remains stationary due to dead motor.	BatterySensor1: Charged LaserInterface1: Moving RFlexEncoderGatherer: Moving	Wheel1 is STOPPED Motor1 is FAILED rFlexCommander1 is GO All components are OK	Wheel1=DRIVING	No possible recovery

Figure 19. Results of Dead Motor Scenario Testing

8.2.6 Discharged Battery – from rest

In this simple scenario, the battery is in a critical state (‘flat’). It is assumed that when the battery is in this state, the motors cease to function although the electronic systems continue as normal, enabling Titan to function.

The goal in this case is for the rover to be driving. The results are shown in Figure 20. Diagnosis in this case is simple as the battery sensor detects the state of charge of the battery directly.

Titan does in fact estimate the state of the rover correctly, and determines that no recovery is possible.

8.2.7 Discharged Battery – While Driving

This scenario is exactly the same as the previous one, except in this case the battery reaches a critical level of charge while the rover is driving. As before, the goal is for the rover to be driving. The results are shown in Figure 21.

It can be seen from the results that Mode Estimation believes that the final state of the system is very different from the actual state. The reason for this becomes apparent upon analysis of the model. Before the battery failure, the rover is in the state where the motor is moving, the wheel is driving and the battery is charged. After the failure, in the real rover, the motor is stopped, the wheel is stopped and the battery is flat. Mode Estimation, however, thinks that the wheel is slipping whilst the battery sensor and the encoder have both failed.

In both the wheel and motor component models, the transitions between DRIVING and STOPPED and ON and OFF respectively have guards which require the command to the overall system to be the ‘stop’ command. This is required for Mode Reconfiguration to work properly, as discussed in section 9. Because of these guards, the ‘desired’ transitions from DRIVING to STOPPED and from ON to OFF when the battery fails cannot happen, since no ‘stop’ command is issued.

Instead, Mode Estimation assigns the most likely set of consistent modes; however to be consistent with the observations this requires a number of component failures. This results in a mode estimate which is far from reality. This is a fundamental limitation of the Titan system which is explored in section 9.

Description	Observations	State Estimated by Titan	Requested Goal	Command from Titan
Initialise with rover stopped. Battery is flat.	BatterySensor1: Flat LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF Battery is FLAT All other components are OK	Wheel1=DRIVING	No possible recovery

Figure 20: Results from Dead Battery (from rest) Scenario

Description	Observations	State Estimated by Titan	Requested Goal	Command from Titan
Initialise with rover stopped. All components are ok.	BatterySensor1: Charged LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is STOPPED Motor1 is OFF Battery1 is CHARGED All components are OK	Wheel1=DRIVING	Send GO command to rFlexCommander1
Command rover to move. Rover moves successfully.	BatterySensor1: Charged LaserInterface1: Driving RFlexEncoderGatherer: Driving	Wheel1 is DRIVING Motor1 is ON Battery1 is CHARGED All other components are OK	Wheel1=DRIVING	No command necessary
Battery reaches critical level. Motor cannot continue to operate. Rover stops.	BatterySensor1: Flat LaserInterface1: Stopped RFlexEncoderGatherer: Stopped	Wheel1 is SLIPPING Motor1 is ON Battery1 is CHARGED Encoder1 is FAILED BatterySensor1 is FAILED rFlexCommander1 is GO LaserInterface1 is OK	Wheel1=DRIVING	

Figure 21: Results from Dead Battery (while moving) Scenario

8.3 Integration with ATRV Testbed

8.3.1 Introduction

In order to demonstrate the capability of the model and Titan on real rovers, the code was integrated with the ATRV testbed. Once this had been carried out, certain scenarios could be tested on the rovers themselves.

8.3.2 Integration

Titan Software

It was desired to run the Titan software on the rover's onboard PC. This required an extensive update of the software previously installed on the rover, which was carried out by the project TA's. The user could interface with Titan via a wireless SSH connection made to a nearby PC. On this PC commands could be sent to Titan, and diagnosis results from Titan could be seen.

Observations

The observations from the laser interface, battery monitor and wheel encoders were available. In order that Titan could use these observations, a number of interfaces had to be written. These mapped intervals in the continuous range of sensor values to discrete variables in Titan, such as 'obs-moving' or 'obs-stopped'. In addition the failure of the laser interface could be detected. These interfaces were developed by the project TA's.

Commands

No direct interface was made between the commands suggested by Titan running on the rover and the commands issued to the rover. The link here was made by the user, who commanded the rover in a certain way depending on the command suggested by Titan.

This method was in fact necessary to allow successful testing and to prevent damage to the rovers. Before the final link can be made, it is necessary to extend the capabilities of the diagnosis and recovery system as well as handling a number of additional interface issues.

8.3.3 Results

The following scenarios were tested on the rover.

a) A simple slipping and recovery scenario
b) A laser hardware failure scenario

In both cases Titan behaved as it had in off-line testing. The results were recorded on video and are included on the accompanying CD-ROM.

8.4 Conclusion

A number of scenarios were devised and tested in order to demonstrate the strengths and weaknesses of the Titan system and the rover model design.

It was shown that the system was able to carry out effective diagnosis and recovery in the slip and slide scenario, which was the overall aim of the project. The capabilities of Titan and the model were also shown in other relatively complex scenarios. Some scenarios, however, revealed inherent flaws in the Titan system. These flaws are discussed in detail in section 9.

It was shown that the diagnosis and recovery system could be integrated with the ATRV test bed, and that the system behaved as off-line tests had predicted.

9 Lessons Learned: Titan's Limitations

The design and implementation of a Titan model to capture the basics of the Rover's dynamics has highlighted several significant shortcomings in Titan's capabilities. Each of these problems is discussed in general terms below, allowing Titan's limitations to be stated explicitly.

9.1 Mode Reconfiguration

Titan's Mode Reconfiguration module is used to obtain the commands required to cause transition to a system state that includes the specified goal modes.

In the following discussion, it is convenient to introduce the requirement that all transition guards are conjunctive clauses. The theory underlying the Titan software includes no such restriction, so generality is maintained if multiple transitions are permitted between a given pair of modes.

Transition guards may include both command and dependent variables, but Mode Reconfiguration's influence is limited to the suggestion of *commands* to achieve the goal state. For this reason, a goal state must necessitate only transitions in which all requirements on dependent variables are already met. Mode Reconfiguration then returns the command values required to complete the satisfaction of the transition guard. Since the persist transition of mode from which the transition takes place is guarded by the negation of the required command, the command suggested by Mode Reconfiguration will force a change of mode.

Consider, however, the case where the goal state requires a transition that includes no command variables. In this case, if the requirements on the dependent variables are already met, Mode Reconfiguration returns `(1)`, indicating that whilst the transition is possible, nothing need be done to make it take place. However, the guard on the persist transition of the starting mode is also met, so the desired transition cannot be forced. Such transitions are therefore beyond Titan's control and Mode Reconfiguration is unable to issue commands to guarantee transition to the goal state.

It is meaningless for the user to specify a fail-mode as a goal and if all goals are ok-modes, Titan will select a goal state that does not include any fault-modes. Therefore, to avoid this issue, we require only that transitions to *nominal* modes include at least one command variable.

Mode Reconfiguration's behavior is logical, but becomes problematic when we consider the entire system. As mentioned in section 5.2, if the transition to the goal state involves *any* mode transitions with guards that do not include command variables, Titan is liable to deduce that the entire system is out of its control. This is because Mode Reconfiguration maintains a concept of the system's "chain of command", that is the pathway of information flow. When looking for the command required to reconfigure the system, it considers components in the order in which they appear on this chain. If a transition without a command variable guard is considered first, then Titan will deduce that the command required to "trigger" the change of state is `(1)`, that is "do nothing". To maintain generality therefore, these problematic transitions must be avoided in all components.

Mode Reconfiguration requires that the guards on all transitions to nominal modes contain at least one external command.

This fundamental requirement is the reason why the wheel component does not include transitions from *SLIPPING* to *DRIVING* and from *SLIDING* to *STOPPED*. Whilst unlikely in practice, both transitions are physically possible and it would seem obvious to model them with an unguarded transition with a low probability. However, to do so would violate the modeling rule derived above. When faced with a goal of *DRIVING* for the wheel mode, Mode Reconfiguration will suggest no action for recovery from the *SLIPPING* mode. This is indeed logical; there is indeed a finite probability that the wheel will spontaneously regain traction. However, Titan's usefulness would be greatly improved if Mode Reconfiguration were able to deduce that a *guaranteed* path, via the *STOP* mode, also exists, and suggest commands to follow this recovery trajectory instead.

9.2 Conflicts Between Mode Estimation and Mode Reconfiguration

For a given a system state, the mode transition probabilities supplied in the Titan model determine the *a priori* probabilities for transition to all possible subsequent states. Titan's Mode Estimation module calculates the *posteriori* probabilities for transition to the states that satisfy a given set of commands and observations.

When evaluating the guard constraints on the mode transitions required to reach a potential subsequent state, Mode Estimation evaluates dependent variables at the *current* state. As a result, transitions guarded by constraints met only in the *subsequent* state cannot be taken.

Some systems, such as the Rover's dynamics, exhibit a clear flow of information along certain "chains" of system components. It is therefore most intuitive to model these systems using components linked from output to input. That is, the dependent variables defining a component's modes are used as guards on transitions in other components. Due to the above result, however, Mode Estimation is unable to reason correctly on models of this type. Instead, components must be linked by compatibility constraints between their dependent variables.

Mode Estimation requires components to be connected by compatibility constraints between the dependent variables that define their modes.

This modeling constraint appears to be fairly insignificant when considered in isolation. However, its conflict with the constraint imposed by Mode Reconfiguration seriously limits Titan's capabilities.

Specification of goal modes requires that at least one component has distinct nominal modes. In addition, components with probabilistic transitions guarded by constraints that do not include command variables are required to represent any transition that is not commanded, such as any kind of failure. Consider a system with these features, at a time-step at which no command is issued. Transitions to un-modeled (and therefore unwanted) failure modes aside, a set of observations may force a component to undergo a probabilistic transition to a fail-mode. This in turn forces other components to change mode in order to maintain compatibility (*due to the ME constraint*). Whilst these other components may have compatible nominal modes, the transitions to these modes cannot take place because no command was issued (*due to the MR constraint*). As a result, these components are forced to enter failure modes, giving unexpected, undesired and unrealistic operation.

For example, consider the Rover system model, which includes a battery component with an ok-mode *OK* and a fault-mode *FLAT*. Assume the Rover is initially driving with all modes nominal and that no un-modeled failure occurs. If the battery sensor detects low charge, then the battery must enter the *FLAT* mode, even if no command is issued. As a result of compatibility requirements, the motor cannot remain in its *ON* mode and this in turn prevents the wheel from remaining in its *DRIVING* mode. The obvious "correct" operation is that the motor enters its *OFF* mode and the wheel its *STOPPED* mode. However, these transitions are to nominal modes and are therefore guarded, so cannot be taken unless a `cmd-stop` is issued. As a result, both the motor and wheel are forced to enter their *FAIL* modes and the model has failed to capture the true behavior of the system. This situation was demonstrated in section 8.2.7.

9.3 Disjunctive Transition Guards

Titan is unable to handle disjunctive transition guards and instead follows the logic mentioned above, whereby each transition guard is presumed conjunctive and multiple transitions are permitted between pairs of modes. However, the current version of Titan considers only one of these transitions, making representation of disjunctive guards impossible. This means that no transition between a pair of modes can result from more than one command. There are obvious cases in which this behavior is desired and frequently it is in fact required due to the modeling constraint imposed by Mode Reconfiguration.

Take for example the Rover's rFlexCommander. The most logical implementation of a reset function would involve transitions from all nominal modes to the *STOP* mode, each guarded by `cmd-stop`. This means that the transition from *GO* to *STOP* is effectively guarded by `(or cmd-stop cmd-reset)`. Furthermore, the *ON* to *OFF* transition in the motor and the *DRIVING* to *STOPPED* transition in the wheel require the same guard due to the modeling constraint imposed by Mode Reconfiguration. This limitation prevents a simple implementation of a reset function and this is the reason for the rather obscure controller model developed for the Rover. This model makes use of a rather artificial *INITIALISING* mode, as well as the dependent `online` to avoid the problem.

9.4 Suggested Solutions

The fact that Mode Reconfiguration returns `(1)` when faced with trying to achieve a mode transition whose guard does not contain a command variable is perfectly logical. The problematic requirement on the way in which a system is modeled could be avoided however, if Titan were able to consider the *union* of all required commands. In this case, even if some transitions were beyond Titan's control, it could nonetheless issue those commands required for other transitions. This would allow correct functionality of a model with guards only on transitions between the nominal modes of input components, where other components were linked through compatibility of model variables. This is a very "neat" solution and a model of this type accurately reflects the way in which changes propagate through the physical system.

The modeling constraint imposed by Mode Estimation, however, is more fundamental and significantly more difficult to overcome. For Mode Estimation to reason correctly on models where dependent variables are used in guards on

transitions in other components, it must be able to consider multiple time steps. In order to meet the guard constraint on a given transition, Mode Estimation would have to look for an intermediate system state in which the relevant dependent variables take on the correct values. This leads to an iterative approach and an enormous increase in complexity. But to achieve this, Mode Estimation would have to do the job of Mode Reconfiguration. This would violate the fundamental division of tasks within Titan, and would require a significant redesign of the software.

The limitations caused by the conflicts between the requirements of Mode Estimation and Mode Reconfiguration are best avoided therefore by the change to Mode Reconfiguration proposed above. In addition, this allows the use of the “neat” system model described above, where command variables are present only on the system’s input devices.

Titan’s inability to handle disjunctive guards is purely an issue of implementation. The theory in Mode Reconfiguration handles both disjunctive and conjunctive operators in conditional logic and the change should be reasonably straightforward.

10 Conclusion and Acknowledgements

During this project, we have had to learn how to use Titan and to build models for a complex system, putting into practice and deepening our comprehension of what we had learned in Course 16.413 about propositional logic and model-based systems. We were faced with several successive issues relative to the use of Titan and its limitations, and we had to elaborate strategies and new design approaches to solve these issues and overcome obstacles.

We eventually designed a model that is able not only to estimate and reconfigure the state of the rover following our initial simple slip-slide scenario, but also to diagnose several other internal hardware failures, and to recover from those failures when possible. With the help of the TAs, who provided us with the necessary software interfaces, we were able to test and demonstrate our model on the actual rovers, simulating various failures from which Titan proved to be able to recover successfully.

We would like to thank the TAs for their advice and for sharing with us their expertise on Titan and the rovers: Stano, Seung, Ollie and Brad. Their help proved to be very valuable to understand subtle details about Titan that could explain the issues we were facing. They also enabled us to test our model directly on the rovers by providing the interfaces between Titan and the on-board sensors.