YAP – Yet Another Planner: User's Manual

for Version 0.1 (demo, released: June 2017)

Mario Gleirscher
Technical University of Munich

August 7, 2017

Abstract

YAP is a software to demonstrate state space exploration for risk analysis and run-time mitigation in dependable autonomous machines. This document provides a brief introduction into the usage of YAP.

About This Working Document:
This manual and guide describes a version of YAP which is under development and part of ongoing scientific research. Passages marked with "Experimental!" indicate incomplete features or features known to be flawed. A more recent version of this document might be available at http://gleirscher.de.

Contents

Chapter 1

What is Yap and What Can It be Used For?

YAP is on its way to become a tool for the modeling, analysis, design, and synthesis of *strategic safety controllers*. The current version of YAP can demonstrate state space modeling, exploration, and shaping as well as symbolic simulation. The objective of YAP is to support engineering, design, and development steps transforming input from hazard analysis into strategic safety controllers. Furthermore, we aim at bridging the gap between safety goals and control applications employing highly automated and autonomous hybrid, adaptive, and model-predictive control.

1.1 Who is Supposed to Use Yap?

YAP is probably best suited to be used by, e.g. *systems or requirements engineers, risk analysts, safety or assurance engineers, and control engineers* having to deal with hazard identification and risk assessment (HARA) and the design, development, and assurance of countermeasures, particularly, in the engineering and assurance of safety controllers for highly automated and autonomous machines. In other words, YAP might be used by engineers responsible for

- the assurance of safety-related properties of
- hazard analysis and risk assessment of
- developing run-time mitigation planners built into

dependable machines under highly automated or autonomous control.

1.2 Some of Yap's Underlying Principles

Multi-Paradigm Analysis. YAP provides a framework to perform risk analysis in a top-down and bottom-up manner. Regarding the breakdown of the con-

trol loop into items, YAP is inspired by top-down refinement as used in methods such as, e.g. B [1], VDM [11], or Z [12]. However, regarding the identification and analysis of causal factors YAP allows to go forward or backward in the causal chain, that is, starting with *root causes* or with *near-mishaps*.

Simplicity, Agility. YAP's objective is to keep the modeling as simple and abstract as possible and add information about the control loop and causal factors ad-hoc or on-demand along with the steps of risk analysis.

Scalability. By allowing a high level of abstraction, YAP aims at supporting the modeling of complex control loops, i.e., processes and controllers responsible for obeying the given control laws.

1.3 About Yap, Licensing, and Acknowledgments

YAP is being developed and maintained by Mario Gleirscher. It represents results of PhD and postdoctoral research at the Department of Informatics of the Technical University of Munich (TUM)[1] and in cooperation with leading companies of the German automotive and software industry. First ideas on the modeling concepts employed in YAP have been published in [7] as well as in follow-up papers [10, 8]. Preliminary concepts and algorithms were investigated between 2012 and 2016, YAP's core development, however, has started in 2017.

Acknowledgments. YAP's features have been inspired by many discussions with safety engineers from German car makers and suppliers. In addition, I would like to thank my colleague Stefan Kugele who has accompanied me with previous work on YAP and its underlying concepts. After all, I am highly grateful for having been able to enjoy several years of fruitful, encouraging, and comfortable working environment at my faculty at TUM. TUM is a great place to do science and teaching.

[1] See http://www.in.tum.de.

Chapter 2

Yap in a Nutshell

This section provides a very compact guide to trying out YAP.

2.1 Getting a Copy of Yap

Currently, you can obtain YAP as a ZIP package via the URL:

http://gleirscher.de/dl/yap-demo.zip

This package includes two folders:

- `examplefiles`: a bunch of example files to be used with this manual, and
- `outputfiles`: a folder available for the storage of the output resulting from the commands mentioned in this manual.

2.2 Installation and Prerequisites

YAP was developed and tested with:

- Java Run-time Environment ≥ 1.8 — for all core features
- GraphViz (DOT) ≥ 2.38 — for export of risk graphs
- LaTeX (pdfLaTeX) with package `ctable` — for export of situation traces
- LINUX operating systems (e.g. Ubuntu 17.04)

2.3 First Steps in Usage

The following list provides a few easy steps to run YAP and get acquainted with its command line interface (CLI):

1. Get command line help with

   ```
   java -jar yap.jar --help
   ```

   For details on the CLI options used below, see Section 2.4 and Table 2.1.

2. To parse and output the example situation "start" and corresponding hazard models, type

   ```
   java -jar yap.jar -c examplefiles/os-start.yap \
        -o outputfiles/os-start.dot \
        -f latex -v -l 3 -s -m --simulate random
   ```

3. To run a simulation with the settings defined in the file `os-start.yap`, type

   ```
   java -jar yap.jar -c examplefiles/os-start.yap \
        -o outputfiles/os-start.dot \
        -f latex -v -l 1 -s --simulate random
   ```

4. To run the same simulation and output reduced risk graphs for the whole simulation run, type

   ```
   java -jar yap.jar -c examplefiles/os-start.yap \
        -o outputfiles/os-start.dot \
        -f latex -v -l 1 --simulate random
   ```

5a. To transform the risk graph for the situation "leaveParkingLot" into PDF, given this situation was visited during simulation, type

   ```
   dot2tex --autosize --figpreamble="\\large" -c -f tikz \
        -t raw -o outputfiles/os-start-2-leaveParkingLot.tex \
        outputfiles/os-start-2-leaveParkingLot.dot

   pdflatex  -output-directory=outputfiles/ \
        outputfiles/os-start-2-leaveParkingLot.tex
   ```

5b. To transform the simulation run (currently, a LaTeX table) into PDF, type

   ```
   pdflatex -output-directory=outputfiles/ simrun-start.tex
   ```

Have fun trying out Yap!

Short	Long Option	Description	Default
`-c`	`--config <FILE>`	Use configuration from file or path `FILE`.	
`-f`	`--format plain|latex`	Generate either LaTeX or plain DOT output.	`plain`
`-h`	`--help`	Show help on command line parameters.	
`-l`	`--log <N>`	Log level `N` > 0 creates log file `FILE.log`, where `1` / SEVERE … `3` / ALL.	`1`
`-m`	`--model`	Show model parsed from `FILE`.	
`-o`	`--output <FILE>`	Direct DOT output into file or path `FILE`.	
	`--simulate initial`	Plan only for initial operational situation.	`initial`
	`--simulate random`	Run multi-step simulation by randomly resolving non-determinism.	
`-s`	`--statistics`	Show information about risk structure generated from the model in `FILE`.	
`-t`	`--taxonomy`	Generate LaTeX taxonomy of endangerments and mitigations, cf. Figures 3.2 to 3.4 and Table A.1.	
`-v`	`--verbose`	For option `-o`, add details to nodes and edges (e.g. embodiment) of the risk graph. Produce more detailed output with the options `-f`, `-m`, `-s`.	
	`--version`	Show version information.	

Table 2.1: Switches available via the command line interface of YAP.

2.4 The Yap Command Line Interface

In general, you can call YAP via

```
java -jar yap.jar -c <FILE>
```

to write results to the standard output. In addition, the call

```
java -jar yap.jar -c <FILE> -o <OUT>
```

writes results into the output file `OUT`. YAP's CLI accepts the switches listed and explained in Table 2.1. Note that the `--simulate` switch is still under development.

Experimental!

Chapter 3

Yap's Input: Situations, Loop, and Hazards

For an appropriate conduct of its analyses, YAP requires information about

- relevant operational situations (Section 3.1),
- the control loop (Section 3.2), and
- relevant hazards (Section 3.3).

For the following, we will assume to have an initially empty file called `example.yap`. Using this file, you can build your own YAP script while following the examples discussed in this guide. In general, a YAP script file can contain the following fragments or compound directives:

```
1 [ Settings { <Body> } ] # single line comment

3  OperationalSituation "situationName" { <Body> }

5 [ ControlLoop "loopName" for "situationName" { <Body> } ]

7 [ HazardModel for "situationName" { <Body> } ]
```

Note 1 *For a simple way of describing the* ***model syntax****, we use* `<Identifier>` *to denote structured non-terminals,* `<A|B>` *to denote choice among* `A` *and* `B`*,* `[A]` *to denote that* `A` *is optional, and* `[A]`* *that* `A` *can be occurring arbitrarily often (incl. not). Below, we also use* BACKUS-NAUR *form rules introduced with "*`::=`*".*

Furthermore, note that YAP at the moment can only handle single line comments introduced with "`#`". Below, we will first discuss the directives `OperationalSituation`, `ControlLoop`, and `HazardModel`. The `Settings` part will be discussed later in Section 4.2.

Most of the examples discussed in this manual focus on the domain of *highly automated and autonomous driving*. However, YAP is intended to be used in other application domains as well.

3.1 The Operational Situation Model

In a first step, YAP has to be provided with an abstraction of the control loop representing the *processes* (also: activities) running (or performed) in this loop. In YAP jargon, this abstraction is called *operational situation (OS) model*. Such a model can consist of one or more *operational situations*. For the classification of parts of the controlled process, we will use the term *aspect*. The partitioning of processes into aspects or OSs usually requires good domain and expert knowledge. It is out of the scope of this manual to dive into the details of this step. However, OSs might be split by:

- the concurrent and sequential processes comprising the whole loop, e.g. "supply power", "driving", "operate vehicle." These processes usually determine the *functionality* of a system [4, 3].

In the file `example.yap`, the directive

```
1 OperationalSituation "situationName"
{
3     <Body>
}
```

would declare an OS labeled with `situationName`. `Body` describes the context of this operational situation and can be empty. In `Body`, you can use the following directives:

- `include`: for
  - **concurrent composition** of activities performed or conductible in an operational situation,[1]
  - **inheritance and reuse** of model information from other YAP files, e.g. YAP model libraries.
- `successor`: for **sequential composition** of activities performed in an operational situation.
- `initialState`: for specifying the *risk state* from which the construction of risk structures (Section 4.1) and mitigation planning will take place. *default:* 0

```
OperationalSituation "situationName"
2 {
  [ include "situationName"; |
```

[1] This is supposed to be the counterpart to parallel composition of MARKOV decision processes (MDP).

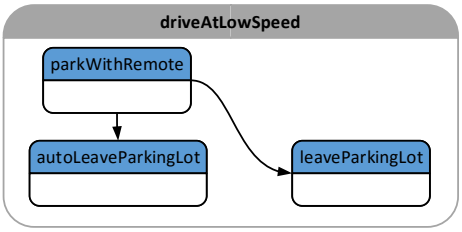

Figure 3.1: Alternative representation of the controlled process declared in Example 1.

```
    successor "situationName"; ]*

  [ initialState (cfId:phase[, cfId:phase]*); ]
}
```

The `cfId:phase` pairs have to refer to an existing causal factor identifier `cfId` (Section 3.3) and use `phase ::= <0|a|m|_>` to specify the initial phase for the associated causal factor (Section 4.1).

Currently, YAP will accept **one** operational situation per file. Note that for each identifier `situationName`, YAP expects a file "`os-situationName.yap`" to exist in the current or corresponding directory.

Example 1 (Modeling the Controlled Process) *The following script applies the directives* `include` *and* `successor`*:*

```
OperationalSituation "parkWithRemote"
{
    include "driveAtLowSpeed";

    successor "autoLeaveParkingLot";
    successor "leaveParkingLot";
}

ControlLoop "myRoboCar" for "parkWithRemote" {}

HazardModel for "parkWithRemote" {}
```

Figure 3.1 shows an alternative representation of this YAP *script fragment using SysML state charts.*[2] *We will discuss the other directives occurring in the situation model below and in Section 4.2.*

3.2 The Control Loop Model

In the next step, YAP can process a model of the *control loop*. In `example.yap`, this can be declared by

[2] A widely used diagram type of the Systems Modeling Language, see [6].

```
ControlLoop "loopName" for "situationName"
{
    <Body>
}
```

For scalability, YAP aims at a high level of abstraction. Hence, in `Body`, you can use the following directives to specify the *items* of the control loop and various kinds of *attributes and relationships between these items*:

- `alias`: for providing more meaningful *item names*,
- `partOf`: for specifying that *an item is a part of another item*,
- `poweredBy`: for specifying that *an active item is provided the required energy by another item* playing the role of an "energy source." Experimental!

The control loop identified by `loopName` is the top-level item and every other item is basically part of the control loop.

```
ControlLoop "loopName" for "situationName"
{
  [ itemId [ alias "itemName" ]
           [ partOf itemId ]
           [ poweredBy (itemId[, itemId]*) ]; ]*
}
```

Note 2 *In* YAP *script, generically, we specify* attributes *by*

```
entity attributeName [attributeValue] ';'
```

and relationships *by*

```
entity relationshipName entity ';'
entity relationshipName '(' entity [',' entity]* ')' ';'
```

where, for ***attributeValue****, string literals are bracketed by "…", e.g.* `"a string"` *and numbers as is by, e.g.* `123`. ***attributeName*** *and* ***relationshipName*** *refer to predefined* YAP *script keywords. Furthermore, entity references such as, e.g.* ***itemId****, have to be strings without white-space characters.*

Example 2 (Control Loop Fragment for "supplyPower") *The following script snippet shows that the control loop fragment for the aspect "supplyPower" consists of an item* ***Pwr*** *referring to the primary energy supply of the controller.* ***Pwr*** *is supposed to be a physical part of the item* ***Ve****, the vehicle.* ***Bat*** *refers to a battery as the alternative energy source. Finally, the script declares an item* ***Ctr*** *to be powered by at least one out of the two resources* ***Pwr*** *and* ***Bat****.*

```
ControlLoop "supplyPower" for "supplyPower"
{
    Pwr alias "PrimaryEnergySource"
```

```
4       partOf Ve;
    Bat alias "Battery"
6       partOf Ve;
    Ctr poweredBy (Pwr,Bat);
8 }
```

3.3 The Hazard Model

At the core of YAP is the hazard model. We have to specify *causal factors*—stemming from hazard identification and risk analysis—and, if applicable, their relationships. For sake of simplicity, we use the terms hazards and causal factor as synonyms.

In our file `example.yap`, the hazard model is introduced through

```
HazardModel for "situationName"
2 {
    <Body>
4 }
```

In `Body`, you can specify the nature of the causal factors chosen to be relevant in the OS `situationName` by using the following directives:

- `alias`: for specifying more meaningful *causal factor names*,
- `activatedBy`: to specify that a causal factor is **activated** by a specific **action** performed by a specific **item**,
- `mitigatedBy`: to specify that a causal factor is mitigated by a specific **action** performed by a specific **item**,
- `causes`: to specify that the **activation** of a causal factor is **propagated** and activates other causal factors,
- `denies`: to specify that the **activation** of a causal factor **denies** the activation of other causal factors,
- `requires`: to specify that the activation of a causal factor requires other causal factors to be **activated in advance** or simultaneously,
- `excludes`: to specify that the activation of a causal factor **superposes** or **invalidates** another causal factor,
- `direct`: to specify that a causal factor can be **completely mitigated**, intentionally at run-time, and

Experimental!

- `offRepair`: to specify that a causal factor can only be (completely) mitigated by putting the **loop out of order**.

Further constraints are planned to be included. Below, we will see a few examples of how to use these directives.

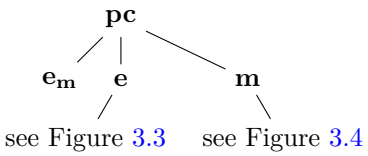

Figure 3.2: Taxonomy of actions; symbols are described in Table A.1 in Appendix A.1.

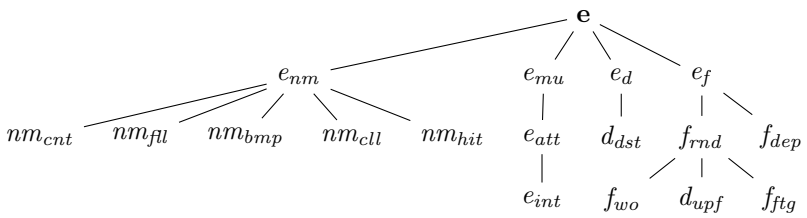

Figure 3.3: Taxonomy of endangerments; symbols are described in Table A.1 in Appendix A.1.

```
HazardModel for "situationName"
{
  [ cfId [ alias "causalFactorName" ]
         [ <activatedBy|mitigatedBy> (actionSpec[, actionSpec]*) ]
         [ <causes|denies|requires|excludes> (cfId[, cfId]*) ]
         [ <direct|offRepair [timeLimit]> ]; ]*
}
```

The directive `actionSpec ::= actionClass[.itemId]` allows for the declaration of …

- `activatedBy`: …endangerments of class `actionClass` from Figure 3.3, and
- `mitigatedBy`: …mitigations of class `actionClass` from Figure 3.4,

performed by the item `itemId`.

Example 3 (Yap Script for the Aspect "supplyPower") *Based on Example 2, Listing 3.1 shows the corresponding* Yap *script. The hazard model specifies the following causal factors:*

- *`lowOrNo-Fuel` (F) describes states where a vehicle will soon be running out of fuel or has already run out. The directive `offRepair` abstracts from the usual practice that the vehicle has to be taken or pulled to a petrol station and that refill is taking place when the vehicle is out of operation. It also specifies that we do not consider refill during operation.*

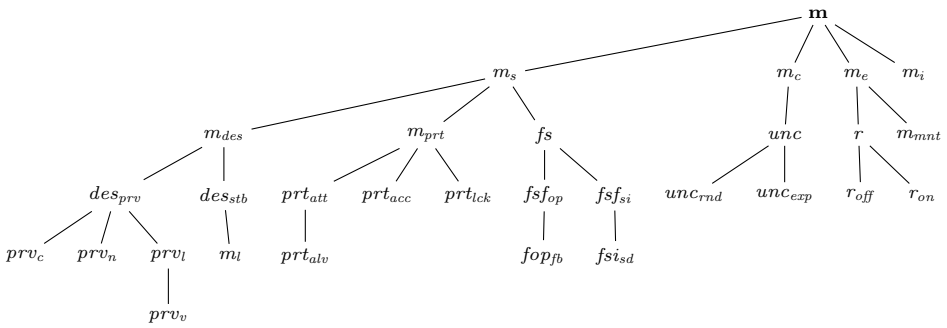

Figure 3.4: Taxonomy of mitigations; symbols are described in Table A.1 in Appendix A.1.

- `lowOrNo-Energy` *(E) describes states where, for some unspecified reason, the vehicle controller might not or is not supplied with sufficient electric power. "*`activatedBy (FAIL.Pwr)`*" specifies the endangerment* `FAIL` *embodied by the item* `Pwr` *which is supposed to activate E. "*`mitigatedBy (FALLBACK.Bat)`*" specifies that we decided to provide a mitigation* `FALLBACK` *performed by* `Bat`*. Finally,* `offRepair` *describes that this causal factor can only be completely reset after putting the vehicle out of operation.*

- `lowOrNo-Battery` *(B) specifies states where battery-based energy supply is soon ceasing or has ceased to work. "*`requires (E)`*" describes that B can only occur in case of failure of the primary energy supply, that is, E. This abstraction incorporates two assumptions: first, the battery is continuously charged by* `Pwr` *and, second, in this specific example we do not consider other causes of B to be relevant, such as, e.g. a broken battery or wire.*

Example 3 indicates how hazard analysis and risk assessment techniques such as HazOp and fault-tree analysis (FTA) [5] can deliver information to be coded into YAP models: The application of the `requires` constraint shows how important it is to align FTA with what is introduced into a hazard specification for YAP. Furthermore, note that the causal factors F, E, and B share the common prefix "lowOrNo" which is a result of applying the guide-words "too low" and "no" to the item `Ctr` of the vehicle's control loop fragment `supplyPower`.

Listing 3.1: YAP script for the aspect "supplyPower."

```
OperationalSituation "supplyPower" {}

ControlLoop "supplyPower" for "supplyPower"
{
    Pwr alias "PrimaryEnergySource"
        partOf Ve;
    Bat alias "Battery"
        partOf Ve;
    Ctr poweredBy (Pwr,Bat);
}

HazardModel for "supplyPower"
{
    F   alias "lowOrNo-Fuel"
        offRepair;

    E   alias "lowOrNo-Energy"
        activatedBy (FAIL.Pwr)
        mitigatedBy (FALLBACK.Bat)
        offRepair within (30min);

    B   alias "lowOrNo-Battery"
        requires (E);
}
```

Chapter 4

Yap's Output: Risk Structures

After having modeled causal factors for a specific control loop in a specific operational situation (see, e.g. Listing 3.1), we can use YAP to generate what we call a *risk structure*.

For this, we run YAP for the situation "supplyPower" using the command

```
java -jar yap.jar -c examplefiles/os-supplyPower.yap \
     -o outputfiles/os-supplyPower.dot \
     -f latex -v --simulate initial
```

and get a *risk graph* as an output. To transform this graph into PDF, we use the commands

```
dot2tex --autosize --figpreamble="\\large" \
     -c -f tikz -t raw -o outputfiles/os-supplyPower.tex \
     outputfiles/os-supplyPower.dot

pdflatex -output-directory=outputfiles \
     outputfiles/os-supplyPower.tex
```

After using these commands, we should get a risk structure compliant with the graph shown in Figure 4.1. The file `os-supplyPower.yap` is included in the demonstration package, see Section 2.1.

4.1 Understanding Risk Structures

Now, it is time to establish a brief understanding of the semantics of graphs such as in Figure 4.1. As shown there, a **risk structure** can be represented by a directed and labeled graph. Risk structures are built from composing all the identified causal factors (Section 3.3) relevant in a specific OS (Section 3.1)

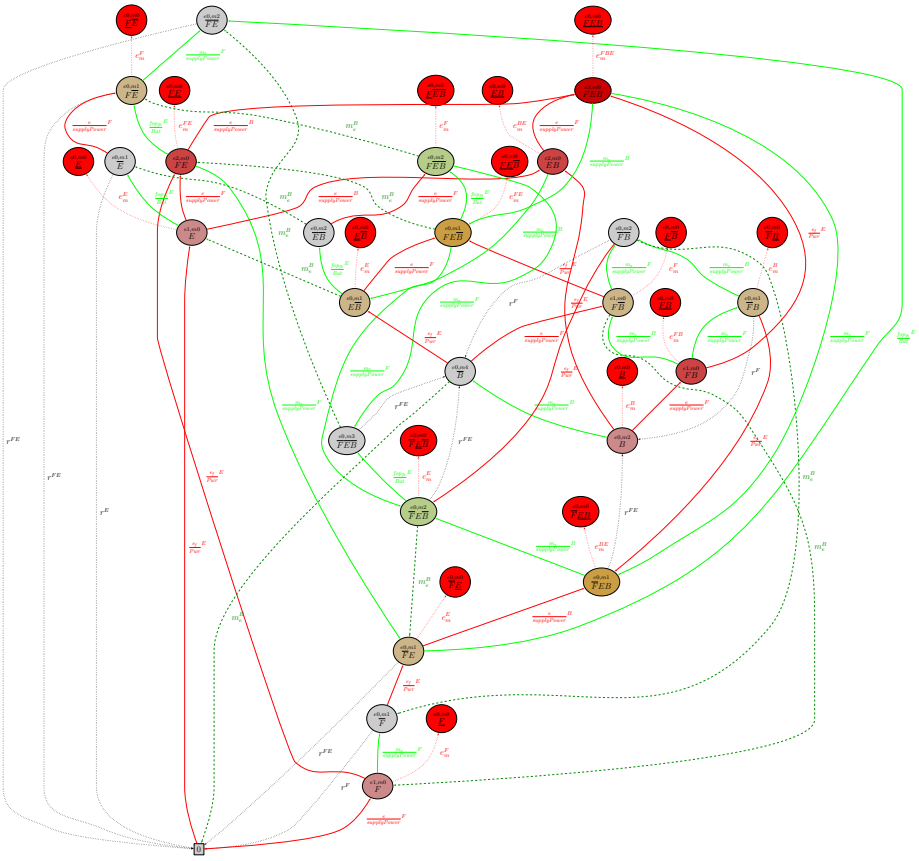

Figure 4.1: Graph generated by YAP with Listing 3.1 and representing the risk structure for "supplyPower."

by using a so-called **phase model**. Particularly, for any causal factor CF, the phase model consists of four phases, as shown in Table 4.1.

The theory underling the construction of risk structures has been elaborated in several publications [9, 8, 10, 7].

Nodes. We distinguish the following kinds of nodes in a risk graph, that is, *risk states* in a risk structure:

- 0: the (locally) "safest state" with none of the causal factors being activated. For sake of brevity, given X, Y, Z are not inactive, we simplify every label "$X0^{CF_i}Y0^{CF_j}Z$" to a label "XYZ" or, if empty, to "0".

- *Undesired events*: labeled with causal factors in either their active or mitigated phases. The superscript eN denotes the number of combined endangerments that have led to this state, mN denotes the number mitigations

Phase Name	Label in Risk Graph	…in Yap Script
inactive	0^{CF}	`0`
active	CF	`a`
mitigated	$\overline{CF}$	`m`
mishap	$\underline{CF}$	`_`

Table 4.1: Phases of the causal factor CF and their labeling.

applied so far, where $N \in \mathbb{N}_0$.

- *Mishaps*: states that represent an unacceptable mishap, indicated in red. The causal factor that is supposed to be the most influencing factor is switched to its mishap phase.

Transitions. We distinguish the following types of edges in a risk graph, that is, *transitions* in a risk structure:

- *Endangerments*: red solid edges denote actions of type `ENDANGER`.
- *Mitigations*: green, dark green, and black edges denote actions of type `MITIGATE`.
- *Mishaps*: red dotted edges denote actions of type `MISHAP`.

Table A.1 in Appendix A.1 provides a corresponding taxonomy of actions. Furthermore, the labels in the graph correspond to the symbols given in the **Symbols** column of this table.

Figure 4.2 provides a smaller and more easily readable example of the elements of a risk structure described so far.

Embodiment. A last element to be mentioned here is the concept of *embodiment* of an action, e.g. the implementation of an endangerment or mitigation. For example, the transition labeled with $\frac{e_f}{Pwr}^{E}$ denotes that the failure e_f activating the causal factor E stems from (electro-physical) behavior embodied by the item `Pwr`. Moreover, the transition labeled with $\frac{fop_{fb}}{Bat}^{E}$ denotes that the fail-operational behavior (*fop*) is realized by a fallback (*fb*) embodied by a managed hand-over to the item `Bat`.

4.2 Settings Controlling Yap's Output

In any YAP script, the compound directive

```
Settings
{
    <Body>
}
```

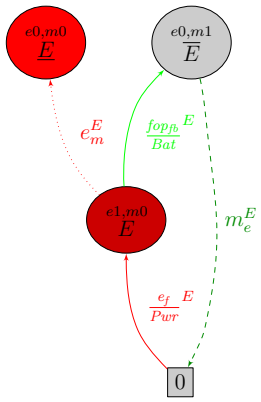

Figure 4.2: Phase model instantiated for the causal factor E from Example 3.

can be used to declare several settings YAP takes into account when performing simulations, generating risk structures, and checking these. In `Body` you can use the following directives:

- `endangermentDepth`: specifies the number of causal factors which can maximally get activated in combination (i.e., sequentially or simultaneously), *default:* 0 (maximum)
- `mitigationDepth`: specifies the number of mitigations which are taken into account for combination (in sequence or simultaneously) when planning a mitigation strategy, *default:* 0 (maximum)
- `simulationLength`: specifies the number of steps to be conducted out of a *scenario* or *run* of a controlled process (Section 3.1), *default:* 0 (infinite)
- `outputDepth`: specifies the depth of the risk graph to be produced as an output. Depth calculation starts at the `initialState` (Section 3.1), *default:* 0 (maximum)

Hence, YAP generates risk structures always from a given specific *initial risk state*. Furthermore, to control the parts of a risk structure to be included in the output, YAP supports the following switches:

- `suppressEndangerments`: if `true`, this switch suppresses all endangerment transitions in the risk graph, *default:* `false`
- `suppressMitigations`: if `true`, suppresses all mitigation transitions in the graph, *default:* `false`
- `suppressLoops`: if `true`, suppresses all transitions leading to phase 0^{CF} of a causal factor CF, *default:* `false`
- `suppressMishaps`: when `true` suppresses the annotation of the risk graph with mishap states. *default:* `false`

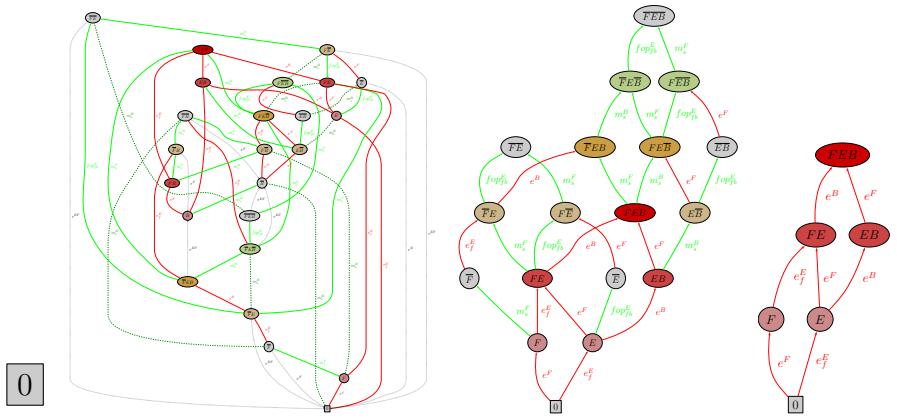

Figure 4.3: Suppressing parts of a risk graph.

In summary, the syntax for the `Settings` directive looks the following:

```
Settings
{
    [ <outputDepth|endangermentDepth|
      mitigationDepth|simulationLength> = natNumber; ]*

    [ <suppressMishaps|suppressEndangerments|
      suppressMitigations|suppressLoops> = <true|false>; ]*
}
```

`natNumber` has to refer to a natural number, i.e., a number in $\mathbb{N}_0$.

Example 4 (Reducing Risk Graphs, Yap's Output) *Given the file* `os--supplyPower.yap`*, e.g. setting* `suppressEndangerments` *to* `false` *in risk state* 0 *would lead to an empty risk structure as there is nothing to which we can apply a mitigation, see Figure 4.3a. As opposed to Figure 4.1, we can omit mishap states by using "*`suppressMishaps = true`*" and get Figure 4.3b. Furthermore, with "*`suppressLoops = true`*" we get Figure 4.3c. Finally, with "*`suppressMitigations = true`*" we get a risk graph reduced to the set of* undesired events *directly reachable by all possible combinations of endangerments or causal factor activations respectively, see Figure 4.3d.*

Chapter 5

Working with Yap

YAP's output can be used to perform two important steps of risk analysis prior to the development of safety controllers and run-time mitigation planners:

- the reduction and shaping of risk structures (Section 5.1) and
- the simulation of scenarios of the controlled process (Section 5.2).

5.1 Reduction and Shaping of Risk Structures

As already introduced in Section 3.3, causal factors can be related via the directives `causes`, `denies`, `requires`, and `excludes`. It goes beyond the scope of this manual to describe further methodological aspects of why, when, and where to use these directives. However, these directives also serve the purpose to simplify the construction of risk structures for run-time mitigation planning.

5.2 Performing Symbolic Simulation

We assume that we have created a bunch of YAP scripts representing the controlled process by using the `include` and `successor` directives (Section 3.1). Next, we type

```
java -jar yap.jar -c examplefiles/os-start.yap \
     -o outputfiles/os-start.dot \
     -f latex -v -l 1 -s --simulate random
```

to run a random simulation starting from the situation "start."

Then, YAP step by step picks a random successor until the `simulationLength` is reached and, for each step, constructs a risk structure. Currently, for demonstration purposes, YAP randomly picks a risk state to jump to when performing the next simulation step. In our case, YAP starts with the situation described in the file `os-start.yap`.

Experimental!

Table 5.1: Simulation run showing 20 steps. Values in parentheses indicate that, for `os-start.yap`, risk structure generation was omitted.

Step	***Operational Situation***	#CFs	Initial State	#States	#Trans.
1	*start*	(0)	(0)	(1)	(0)
2	*leaveParkingLot*	3	EB	5	5
3	*driveAtL1Generic*	10	$EBOI$	197	452
4	*manuallyPark*	3	FEB	3	2
5	*autoLeaveParkingLot*	3	B	11	14
6	*driveAtL4Generic*	10	$FEOnTOCrA$	638	1564
7	*autoOvertake*	10	$EOWnCCnTOC$	1192	3294
8	*driveAtL4Generic*	10	$OnCCnAPnTOCrA$	487	1226
9	*exitTunnel*	10	$BOWnCnTOCrA$	1171	2999
10	*driveAtL4Generic*	10	$OWnCnTOC$	908	2127
11	*driveThroughCrossing*	10	$FOWnC$	1005	2380
12	*driveAtL1Generic*	10	$FEOWCDoL$	68	158
13	*manuallyPark*	3	E	11	12
14	*autoLeaveParkingLot*	3	FE	6	6
15	*driveAtL4Generic*	10	$EOWnCrA$	1803	4760
16	*driveThroughCrossing*	10	$EnCCnTOC$	1075	2940
17	*driveAtL1Generic*	10	$OWnCoL$	762	2029
18	*manuallyPark*	3	FE	6	6
19	*leaveParkingLot*	3	F	7	7
20	*driveAtL1Generic*	10	BOW	362	942

Example 5 (Symbolic Simulation) *Table 5.1 shows a simulation run using the following settings:*

```
Settings
{
    outputDepth = 2;
    endangermentDepth = 1;
    mitigationDepth = 1;
    simulationLength = 20;

    suppressMishaps = true;
    suppressEndangerments = false;
    suppressMitigations = false;
    suppressLoops = true;
}
```

Consider the various risk states jumped into (column "Initial State") and that the endangerment complexity (columns "#States" and "#Trans.") of some of the steps includes the regard of 10 causal factors. This run took about 87 sec.

Please, take a look at Section 7.2 for further comments on implications that, particularly, come along with the *state space explosion problem* [2].

Chapter 6

More Examples

For the time being, I do not provide further examples. However, please, let me know of any successes

- in trying out YAP involving example scripts you crafted or
- in using YAP for some part of your analysis as an engineer in one of the roles mentioned in Section 1.1.

Feel free to send me your YAP scrips (`.yap` files) to the email address given below in Chapter 7.

Chapter 7

FAQ, Troubleshooting, and Limitations

I appreciate any reports about bugs in YAP, illustrative examples, and feature requests. Please, report to

mailto:mario-dot-gleirscher-at-tum-dot-de.

However, please, take into account that, due to further professional obligations, I cannot provide any guarantee on when or whether at all corresponding fixes and extensions will get introduced in YAP.

7.1 Frequently Asked Questions and Troubleshooting

None reported or relevant so far.

Troubleshooting. For some mistakes in using the CLI (though, in more obvious and general cases), YAP will provide you with error messages right on the command line.

For further mistakes in the CLI or in YAP scripts (e.g. model incompleteness and inconsistencies), YAP appends some information about its internal processing (e.g. warnings, error messages, notes about processing steps) to a log file associated with the `FILE` specified via the CLI option `--config`, see Section 2.4. Hence, you will find a file called `FILE.log` in the directory with the corresponding YAP scripts. Furthermore, you can raise the log level using the option `--log` to get more such information.

Experimental!

7.2 Known Bugs and Limitations

None reported or relevant so far.

Limitations. YAP is still in an early stage with *demonstration and proof-of-concept* as its preliminary design goal and, thus, exhibits a number of algorithmic complexity issues that can be reduced, and a number of optimizations in time and memory consumption that have not yet been applied. Fixes to part of these issues are already known and on my schedule.

Appendix A

More Technical Details

This section of the manual provides details about YAP required for its more in-depth usage.

A.1 Taxonomy of Action Classes

Table A.1 provides the currently supported list of *action types*. Please, refer to Section 3.3 to read about the usage of these action types.

Table A.1: Comprehensive list currently supported action types.

`actionClass`	**Symbol**	**Description of Action Class**
`PHASECHANGE`	pc	Class encompassing all action classes.
Types of Endangerments		
`ENDANGER`	e	Class of all endangerment actions.
`MISHAP`	e_m	Mishap actions leading to hazard phases without mitigation.
`FAIL`	e_f	Fault actions modeling transient or permanent random faults.
`DISTURB`	e_d	Disturbance actions, e.g. abrupt perturbation, unforeseen obstacle, sensor noise.
`MISUSE`	e_{mu}	Unintentional and intentional maloperation.
`NEAR_MISHAP`	e_{nm}	Actions exhibiting reducible events, e.g. collisions which can be alleviated.
`ATTACK`	e_{att}	Attack actions, e.g. IT attack, soft or physical attack.
`INTRUDE`	e_{int}	Intrusion actions, e.g. traceable unauthorized access.
`FAIL_DEPENDENTLY`	f_{dep}	Fault actions modeling compound events without modeled causes, e.g. common cause, common mode, single point, multiple point, cascading faults.
`FAIL_RANDOMLY`	f_{rnd}	Fault actions modeling semi-systematic faults, e.g. basic events in fault-trees.

cont'd on next page

Table A.1: Comprehensive list currently supported action types (cont'd).

actionClass	Symbol	Description of Action Class
WEAR_OUT	f_{wo}	Fault actions characterizing wear-out, deterioration, material fatigue, or decay.
FATIGUE	f_{ftg}	Fault actions representing operator fatigue.
UNDERPERFORM	d_{upf}	Fault actions modeling currently unacceptable performance, e.g. delayed execution.
CONTAMINATE	nm_{cnt}	Actions for modeling contamination of areas with hazardous materials.
COLLIDE	nm_{cll}	Actions for modeling collisions of valuable assets.
HIT	nm_{hit}	Actions representing that valuable assets are hit by hazardous objects.
FALL	nm_{fll}	Actions modeling valuable assets falling from hazardous height.
BUMP	nm_{bmp}	Actions representing passive collisions of valuable assets.
DISTRACT	d_{dst}	All kinds of distraction of human operators.
Types of Mitigations		
MITIGATE	m	Class of all mitigation actions, i.e., for hazard treatment
START_MITIGATE	m_s	Class of mitigations not automatically leading to original phase, i.e., initiations of mitigations.
INTER_MITIGATE	m_i	Class of multiple step mitigations.
END_MITIGATE	m_e	Class of completions for partial mitigations.
COMPLETELY_MITIGATE	m_c	Class of mitigations directly leading to original phase.
FAIL_SAFE	fs	Class of mitigations dealing with defect treatment.
DE_ESCALATE	m_{des}	Actions de-escalating a hazardous event or situation.
PROTECT	m_{prt}	Actions preventing from the occurrence of a hazardous event or situation by protection mechanisms, e.g. safety barriers
REPAIR	r	Actions dealing with the repair of a causal factor, e.g. a fault, and its consequences.
DO_MAINTENANCE	m_{mnt}	Actions representing maintenance, e.g. for mitigation of early-stage causal factors.
UNCONTROLLED	unc	For modeling externally, randomly mitigated causal factors, out of the scope of the controller.
EXPECTED	unc_{exp}	Actions modeling expected but uncontrolled causal factors, e.g. passive collisions.
RANDOM	unc_{rnd}	Actions modeling random but uncontrolled causal factors, e.g. passive collisions whose frequency is known.
ATTENUATE	prt_{att}	Actions representing attenuation mechanisms in general, e.g. car airbag.
CONTROL_ACCESS	prt_{acc}	Actions modeling restricted access to a valuable asset, e.g. block access to rooms, IT infrastructure, HMI controls.
INTERLOCK	prt_{lck}	Actions controlling physical access to shared resources like such as rail tracks, road crossings, flight route segments by mechanisms e.g. road traffic lights, train interlocking systems, air traffic control.

cont'd on next page

Table A.1: Comprehensive list currently supported action types (cont'd).

actionClass	Symbol	Description of Action Class
ALLEVIATE	prt_{alv}	Actions encompassing mechanisms for passive safety, e.g. airbag, safety belt, bumper.
MAINTAIN_STABILITY	des_{stb}	Actions representing stabilization mechanisms in the control loop, e.g. ESP, DSC.
PREVENT	des_{prv}	Actions preventing from the occurrence of a hazard event or situation, e.g. highly attentive driver.
LIMIT	m_l	Actions limiting potentially hazardous control actions, e.g. ABS.
PREVENT_CRASH	prv_c	Actions modeling active or preventive safety, e.g. distance control, collision avoidance, metric reach avoid control, emergency braking.
PREVENT_LOSSOFCONTROL	prv_l	Actions reducing risk of de-stabilization by, e.g. maintenance of remote or internal control.
NOTIFY	prv_n	Warning actions, e.g. (digital) road signs for vehicle/driver, warning indicator lights for driver and environment, warnings for pilots.
CHECK_VIGILANCE	prv_v	Mechanisms for vigilance checking, e.g. dead man switch, driver fatigue detection.
REPAIR_OFFLINE	r_{off}	Actions repairing causal factors requiring shutdown of the control loop.
REPAIR_ONLINE	r_{on}	Actions repairing causal factors during operation of the control loop.
FAIL_SILENT	fsf_{si}	Actions representing mechanism for shutting down parts of the controller.
FAIL_OPERATIONAL	fsf_{op}	Actions representing mechanisms for maintaining functionality of the controller by, e.g. redundancy, degradation, fail-over (via many design tactics), hand over to human operator on machine failure, take over from human operator on operator failure.
SHUTDOWN	fsi_{sd}	Actions dealing with systematic shutdown or deactivation, e.g. emergency halt or stop.
FALLBACK	fop_{fb}	Actions representing degradation to a backup component of the controller.

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. 1st ed. Cambridge University Press, June 2010. ISBN: 9780521895569.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, May 2008. ISBN: 9780262026499.

[3] Manfred Broy. "Multifunctional software systems: Structured modeling and specification of functional requirements". In: *Science of Computer Programming* 75.12 (2010), pp. 1193–1214. DOI: 10.1016/j.scico.2010.06.007.

[4] Manfred Broy. "Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures – The JANUS Approach". In: *Engineering Theories of Software Intensive Systems*. Springer, 2005, pp. 47–81. ISBN: 978-1-4020-3532-6.

[5] Clifton A. Ericson. *Hazard Analysis Techniques for System Safety*. 2nd. Wiley, July 2015. ISBN: 9781118940389.

[6] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. 3rd. Morgan Kaufmann, July 2014. ISBN: 9780123743794.

[7] Mario Gleirscher. "Behavioral Safety of Technical Systems". Dissertation. Technische Universität München, Dec. 2014. DOI: 10.13140/2.1.3122.7688.

[8] Mario Gleirscher. "Run-Time Risk Mitigation in Automated Vehicles: A Model for Studying Preparatory Steps". In: *1st iFM Workshop on Formal Verification of Autonomous Vehicles (FVAV)*. EPTCS. submitted. 2017.

[9] Mario Gleirscher and Stefan Kugele. "Defining Risk States in Autonomous Road Vehicles". In: *High Assurance Systems Engineering (HASE), 18th Int. Symp.* Jan. 2017, pp. 112–5. DOI: 10.1109/HASE.2017.14.

[10] Mario Gleirscher and Stefan Kugele. "From Hazard Analysis to Hazard Mitigation Planning: The Automated Driving Case". In: *NASA Formal Methods (NFM) – 9th Int. Symp., Proceedings*. Ed. by C. Barrett et al. Vol. 10227. LNCS. Springer, Berlin/New York, May 16, 2017. DOI: 10.1007/978-3-319-57288-823.

[11] Cliff B. Jones. *Systematic Program Development Using VDM*. Prentice Hall, 1986. ISBN: 0-13-880725-6.

[12] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Englewood Cliffs: Prentice Hall, 1991. ISBN: 0-13-478702-1.