YAP Against Perils: Application Guide and User's Manual[1]

draft for program version 0.7.2+20210704 (built on 2021-07-04)

Mario Gleirscher
Mathematics & Computer Science
University of Bremen, Germany

July 4, 2021

[1]This work was supported by the German Research Foundation (Deutsche Forschungsgemeinschaft) under the grant no. 381212925 and the Lloyds Register Foundation Assuring Autonomy International Programme grant CSI:Cobots.

Abstract

YAP is a research tool for risk modelling—particularly, for exploring and investigating abstract state spaces for situational risk analysis—and for controller synthesis—particularly, the synthesis of controllers for run-time handling of critical events occurring in the operation of highly-automated and autonomous systems. This document provides a guide to the concepts and usage of YAP as well as a technical manual for the tool.

About this document:
This book, guide, and manual describes YAP, a prototypical tool 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 from http://yap.gleirscher.de.

Contents

Abbreviations

AAIP	Assuring Autonomy International Programme	p. 1
BTA	bow tie analysis	p. 7
CE	critical event	p. 7
CLI	command line interface	p. 13
CSP	communicating sequential processes	p. 21
DFG	German Research Foundation	p. 1
DFT	dynamic fault tree	p. 15
DFTA	dynamic fault tree analysis	p. 6
DSL	domain-specific language	p. 6
FMEA	failure mode and effects analysis	p. 6
FTA	fault tree analysis	p. 7
GCL	guarded command language	p. 4
HARA	hazard analysis and risk assessment	p. 5
HazOp	Hazard and Operability studies	p. 6
LOPA	layer of protection analysis	p. 6
LRF	Lloyds Register Foundation	p. 1
LTS	labelled transition system	p. 6
MCSeq	minimal cut sequence	p. 5
MDP	Markov decision process	p. 5
OS	operational situation	p. 18
PCTL	probabilistic computation tree logic	p. 52
pGCL	probabilistic *guarded command language* (GCL)	p. 21
RCA	root cause analysis	p. 7
STPA	System-Theoretic Process Analysis	p. 6
STS	symbolic transition system	p. 8
SysML	Systems Modelling Language	p. 19
TUM	Technical University of Munich	p. 9
UoB	University of Bremen	p. 8
UoY	University of York	p. 9
VDM	Vienna Development Method	p. 7

Chapter 1

What is Yap and What Can It be Used For?

YAP (short for *YAP Against Perils*[1]) is a research tool for the modelling, analysis, design, and synthesis of *strategic safety controllers*. The current version of YAP can demonstrate state space modelling, exploration, and shaping as well as symbolic simulation. One can generate *minimal cut sequences* (MCSeqs), calculate risk state spaces and properties thereof, and synthesise safety controllers for systems given as *Markov decision processes* (MDPs). The objective of YAP is to support engineering, design, and development steps transforming inputs from hazard analysis into strategic safety controllers. Furthermore, YAP seeks to bridge the gap between safety goals and control applications employing highly automated and autonomous hybrid, adaptive, and model-predictive control. However, YAP is not a general-purpose planning framework or tool.

1.1 Who is Supposed to Use Yap?

YAP is best suited to be used by, for example, *systems or requirements engineers, risk analysts, safety or assurance engineers, and control engineers* dealing with *hazard analysis 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 (Figure 1.1). YAP might be used by engineers responsible for

- the assurance of safety-related properties of
- hazard analysis and risk assessment of
- developing safety monitors and mitigation controllers built into

dependable machines under highly automated or autonomous control.

[1] Or *YAP, A Planner* and, formerly, *Yet Another Planner*.

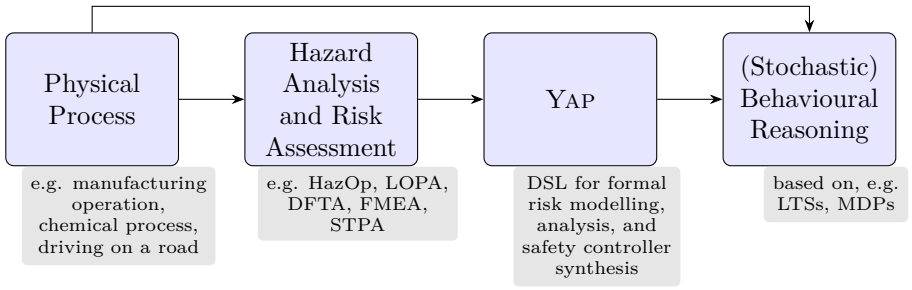


Figure 1.1: Exemplary workflow to be used with YAP

1.2 Some of Yap's Underlying Principles

Safety Constraints. Safe behaviour, whether in sports, on roads, in households, etc. means staying within safety margins or envelopes, keeping key indicators below safety thresholds, not exceeding safety limits, generally, maintaining certain invariants or not violating certain *safety constraints*. Such constraints emerge from past experience (e.g. incidents, accidents) and corresponding risk analyses, carrying the assumption that the likelihood of something bad happening in periods where a constraint is violated is significantly higher than in periods where the system is operated within the constraint.

This notion of safety seems natural, particularly, if we understand risk-averse behaviour as being a good thing in general. Unsurprisingly, this notion has been adopted as a best practice in most safety-critical engineering domains as early as system accidents indicated the causes and their controllability. It was Leveson (2012) who helped the notion of safety constraints gain much more importance in the highly interdisciplinary field of software-based control of critical systems. Leveson (2004) proposed to treat safety as an *emergent property*, in particular, as a behaviour emerging from the interaction of the potentially many agents in a controlled process staying within carefully formulated safety constraints.

Important questions for engineers and operators of a machine are: (i) What are the constraints to be enforced? (ii) How can we enforce or not violate them? (iii) How can we keep violations minimal and short? (iv) How can we verify that an implementation enforces these constraints? Enforcement includes both the *detection* of behaviours near constraint boundaries and the *handling* (e.g. avoidance) of constraint violations ideally followed by the *resumption* of nominal behaviour. In complex systems, the task of formulating a system-level safety constraint will have to be broken down into identifying many local constraints whose interrelationships are reasonably well understood in order to enforce an overall constraint that appropriately represents what safety means for the whole system. Moreover, several constraints may address different aspects of safety and some of these constraints may be related and composed into *compound constraints*.

A particular form of compound constraints, *layered* constraints, is used in

LOPA and *bow tie analysis* (BTA, Ruijter et al. 2016), two risk assessment techniques applicable to systems with several layers of protection. Each of these layers can be associated with a certain *critical event* (CE) (e.g. hazard) and a corresponding safety constraint based on which certain controls or interventions protect the system from escalating the CE to the next layer (or the next higher level of risk). Layered constraints are useful to breakdown risk in complex hazardous processes where several intervention points are realisable, such as in chemical processes, food production processes and supply chains, or air or road traffic management.

Several lines of research follow these ideas, for example, active safety monitoring (Machin et al. 2018) allows one to decompose constraints into discrete layers, and risk-sensitive control (Sanger 2014) estimates and minimises risk on a continuous scale. YAP aims at combining such approaches and supporting the modelling and handling of multiple hazards and the structuring of controller's action sets for concurrent monitoring and mitigation.

Assumption/Guarantee-style Constraints. Often we can safely constrain machine behaviour, and effectively reduce accident likelihood only under certain environmental *assumptions*. This means, the agents[2] responsible for behaving according to the specified *constraints* (i.e., safety requirements) must only do so if the agents not responsible for these constraints behave according to these assumptions. In autonomous systems, this critical responsibility has to be given to the machine but in other systems, it is shared between the machine and the human operator or is even entirely left to operators. Frequently, it is of interest to keep safety assumptions weak so they account for uncertainties (Jackson 2001) and to keep safety requirements strong while the machine remains[3] useful.

Returning to Leveson's emergent properties, safety needs to be verified of the overall machine. Hence, local verification results need to be confidently integrated (Gleirscher, Foster, et al. 2019). YAP provides guidance for structuring risk models as layered constraints, for formalising these models, and for making these models amenable to verification by specialist techniques and tools.

Multi-Paradigm Analysis. YAP allows one to perform risk analysis in a top-down and bottom-up manner. Regarding the breakdown of the control loop into items, YAP is inspired by top-down refinement as used in methods such as, for example, B (Abrial 2010), VDM (Jones 1986), or Z (Spivey 1989). However, currently, YAP's input language is based on abstract LTSs rather than relations, which limits the refinement notions applicable. Regarding the identification and analysis of *risk factors*[4] YAP allows one to go forward or backward in the causal chain, for example, starting with *root causes* or with *near-mishaps*. Risk factors are a generalisation of faults, causal factors, failures, hazards, mishaps,

[2] Machine components and/or human operators, in summary, the *safety or mitigation controller*.

[3] Conflicts between safety and performance frequently complicate machine implementations.

[4] Also called causal factor in *root cause analysis* (RCA) or basic event in *fault tree analysis* (FTA, Ericson 2015).

incidents, and accidents. They can be composed to form risk states (Gleirscher, Calinescu, and Woodcock 2021).

Actions rather than Signal Flow. Modern system safety analysis is heavily based on system *models*. Such models can capture the *structure* or architecture of a system, interaction between components of an architecture in terms of *data or signal flow*. Other models concentrate on representing *behaviour*, as *input/output relations* at the interfaces of the components, as data or control *states and transitions* between such states, and as *action systems or functions* for the specification or generation of such behaviour (Broy 2010).

Widely used techniques for defect, fault, or deviation (e.g. failure logic) analysis usually adopt one or more of these concepts. For example, FMEA and FTA, when applied to control system architectures, are typically given a signal flow interpretation based on the system structure, with the goal of mapping undesired (regions of) output signals (or events) to combinations of internal faults and undesired (regions of) input signals (or events), or vice versa.

YAP is *not* about functional safety. YAP focuses on state-based behaviour and events or actions causing state transitions (Gleirscher 2014). Data flow is abstracted to a large extent, leading to the advantages and disadvantages of *symbolic transition systems* (STSs). One of YAP's core assumptions is that the user (i.e., the analyst or engineer) can associate a measure of risk, danger, or other negativity with (regions of) the considered state space. In practice, this is usually the case for control applications. Readers coming from the functional safety domain may be more familiar with the signal flow perspective.

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

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

In overall, YAP follows a light-weight modelling paradigm as, for example, shown by Lamsweerde (2009) and Letier (2001). To achieve a good level of practicality, YAP's conceptual primitives are inspired by these and related works.

1.3 About Yap, Acknowledgements, and Licensing

YAP is developed and maintained by Mario Gleirscher at the *University of Bremen* (UoB).[5] YAP includes results of postdoctoral and PhD research at the

[5] See www.uni-bremen.de.

Computer Science Department of *University of York* (UoY),[6] at the Department of Informatics of the *Technical University of Munich* (TUM),[7] and from cooperations with leading companies of the automotive and software industry. Preliminary results implemented in YAP are published in Gleirscher (2014) and in follow-up papers (Gleirscher 2017; Gleirscher and Kugele 2017b; Gleirscher 2018). Some ideas and algorithms were investigated between 2012 and 2016, YAP's development, however, has started in Spring 2017, and the formal investigation of the theory underlying YAP is subject of ongoing research (Gleirscher, Calinescu, and Woodcock 2021).

Acknowledgements. The development of YAP was supported by the LRF under the AAIP grant CSI:Cobot and by the German Research Foundation (Deutsche Forschungsgemeinschaft) under the grant no. 381212925.

I would like to thank several anonymous safety practitioners for fruitful discussions and collaborations, having inspired me to develop and enhance YAP. Moreover, I am grateful to my mentors, particularly, Manfred Broy, Radu Calinescu, Ana Cavalcanti, Jan Peleska, and Jim Woodcock, and to my academic colleagues Simon Foster, Stefan Kugele, and Diego Marmsoler for discussions and collaborations on a variety of topics allowing me to use or develop YAP for.

Parts of YAP have been designed and developed with Apache NetBeans, GNU Emacs, other GNU software, and Ubuntu.

[6] See www.cs.york.ac.uk.
[7] See http://www.in.tum.de.

incomplete or otherwise undesired features. Also, I apologise if the documentation is not fully in-sync with the software. Please, let me know if you spot any issues related to YAP or its supportive materials.

Chapter 2

Yap in a Nutshell

This section provides a very compact guide to getting acquainted with YAP and making first steps.

2.1 Installation Requirements and Suggestions

YAP requires and was developed and tested with:

- LINUX operating system (e.g. Ubuntu 17.04, 18.04, 19.10, 20.04), Mac OS (10.15.*), and Windows 10; it probably runs with a variety of older and newer distributions as well
- Java Run-time Environment ≥ 1.8[1] for all core features
  - Linux: e.g. package `default-jre[-headless]`
  - Mac OS or Windows: e.g. directly from Oracle (see https://www.java.com/en/)
- GraphViz (DOT) ≥ 2.38 with `tikz` option for export of risk graphs
- LaTeX (pdfLaTeX) with packages `tikz` and `ctable` for export of graphs and activity traces

For using YAP, the following tools are suggested to be installed:

- Probabilistic model checker (PRISM 4.5 or newer,[2] or a PRISM-compatible tool such as STORM[3]) for using the optimal controller synthesis features
- GNU Emacs 25.2.x for `yap-mode`, a major mode for YAP
- PDF viewer for display in `yap-mode` (e.g. GNOME `Evince`, Zathura)

[1] This is subject to change. YAP 0.5 and newer requires Java 11 or newer.
[2] See www.prismmodelchecker.org.
[3] See https://www.stormchecker.org.

- `pdftk` for compilation of activity traces into a single file

In Mac OS, the package managers **macports**[4] and **homebrew**[5] make GraphViz, LaTeX, GNU Emacs, and PDF viewers available. For Windows, I can recommend **Cygwin**.[6] Please, let me know if you want to help me testing YAP on Mac OS X or Windows.

2.2 Getting and Installing a Copy of Yap

Currently, you can obtain YAP as a

- ZIP (or TAR.GZ) package via the URL

  http://yap.gleirscher.de/dl/yapp_VERSION.zip

  and to be installed in any directory of convenience.

- DEB package to be installed with

  ```
  sudo apt-get install default-jre-headless jarwrapper gawk \
    java-common openjdk-11-jre-headless libsigsegv2 \
    binfmt-support fastjar ca-certificates-java
  sudo wget http://yap.gleirscher.de/dl/yapp_VERSION_all.deb
  sudo dpkg -iG yapp_VERSION_all.deb
  ```

  and de-installed with

  ```
  sudo apt-get purge yapp
  ```

  This package has been tested with Ubuntu 18.04, 19.10, and 20.04. The DEB package contains a man page `yapp(1)` and a Java wrapper script `yapp`.

Both packages contain this manual, a collection of example files, and the main binary `yapp.jar`. The **example files** to be used with this manual are located in the folder

- `examples` of the ZIP package, or
- `/usr/share/yapp/examples` after installation from the DEB package.

Because `/usr/share` is normally read-only for non-root users, it is recommended to copy these files into a directory `~/yapp-examples` and to have YAP generate any output from the commands mentioned in this manual into a folder `~/yapp-examples/output`.

Note 1 *To avoid confusion with the Prolog implementation* ***yap****, YAP packages and executables carry the name* ***yapp*** *for "YAP Against Perils Package." I will need a few more ☕ to resolve this issue at some point.* ☺

[4] See https://www.macports.org.
[5] See https://brew.sh.
[6] See https://cygwin.com.

2.3 First Steps in Using Yap

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

Note 2 *You may invoke YAP by calling* `yapp` *(the wrapper script),* `java -jar yapp.jar` *(the binary; both installed from the DEB package), or* `java -jar yap.jar` *(the binary available from the ZIP package). In the following, we assume that the DEB package was installed and show only the first option.*

1. Follow the advice in Section 2.2 by copying the content of the `examples` directory into `~/yapp-examples/` and change to that directory.

2. Get command-line help with

   ```
   yapp --help
   ```

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

3. To parse the example activity start and to show the parsed and consolidated model, type

   ```
   yapp -m autodrv/start.yap -o output/start.parsed \
        -f latex -v 1 -l 2 -s --simulate random \
        --showmodel raw
   ```

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

   ```
   yapp -m autodrv/start.yap -o output/start.dot \
        -f latex -v 1 -l 2 -s --simulate random
   ```

5. To run the same simulation and output a risk graph for each step of the simulation run, type

   ```
   yapp -m autodrv/start.yap -o output/start.dot \
        -f latex -v 1 -l 2 --simulate random
   ```

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

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

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

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

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

Have fun with Yap!

2.4 The Yap Command-Line Interface

In general, you can call YAP via

```
yapp -m <FILE> [OPTIONS]
```

to write results to the standard output. Alternatively, the call

```
yapp -m <FILE> -o <OUT> [OPTIONS]
```

writes results into the output file `OUT`. YAP's CLI accepts the switches listed and explained in Table 2.1. The options `-v` (verbosity level) and `-l` (logging level) can be useful for model debugging. Features enabled via the `--simulate` switch might change significantly in the future.

Experimental!

In MS Windows, from the `demo` folder of the extracted ZIP package, one can call YAP with

```
C:\jdk13.0.1\bin\java -jar yapp.jar -m .\examples\<FILE>.yap \
  -o .\examples\output\<OUT>.dot
```

Table 2.1: Switches available through YAP's command line interface

Short/long Option	Description	Default
Options		
`-a --activity ACT`	Start simulation from activity `ACT`. By default, the activity to start with will be chosen from the parameter `FILE` provided with the option `-m`.	
`--clear-logs`	Clear log file before start.	`false`
`-d --design monitor-only|single-event|-multi-event-sequential|-multi-event-concurrent`	Choose the basic design of the synthesised controller. Currently, YAP offers the options: monitor-only, single-event, and multi-event-concurrent, explained in more detail in Gleirscher (2020).	`multi-event-concurrent`
`--force`	Override internal sanity checks, e.g. do not check program version when parsing a model.	`false`
`-f --format latex|plain|-prism`	Generate either LaTeX or plain `dot` output. PRISM experimental!	`latex`
`--global-logging`	Store log messages in the global file `/.yap/yap.log` in the user's home directory.	`true`
`-l --log-level 0|1|2|3`	Log level `N` > 0 creates log file `FILE.log`, where `1` / SEVERE … `3` / ALL.	`SEVERE`
`-m --model FILE`	Use configuration from file or path `FILE`.	
`--nosimplify`	do not apply rules for reduction of risk structure complexity beyond the constraints specified in the factor model. Currently, the only rule in place is **mis** (Gleirscher 2018), constructing equivalence classes of mishap states and re-routing transitions to class representatives.	`true`

cont'd on next page

Table 2.1: Switches available via the command line interface of YAP (cont'd)

Short/long Option		Description	Default
-o	--output FILE	Direct output (e.g. dot) into file or path FILE.	output.yap
	--severity	Calculate severity for the whole risk space.	false
-t	--target-model TARGETFILE	where TARGETFILE contains an external (e.g. PRISM) model to be used for synthesis	input.txt
-v	--verbosity-level 0|1|2|3	For option -o, add details to nodes and edges (e.g. mitigation embodiment, state severity) of the risk graph. Produce more detailed output with the options -f, -m, -s. The detail level N > 0 signifies the amount of information produced depending on the context.	0
Commands			
-h	--help	Show help on command-line parameters.	false
	--mincs	Calculate and output all minimal cut sequences from the initial state to all registered mishap factors.	false
	--showlogs no|all|follow	Print all log file contents or follow tail of the log file.	no
	--showmodel no|raw|-activities|-dft	Show model parsed from FILE (raw), DOT graph of activities (activities), or a *dynamic fault tree* (DFT) (dft). Suppress model (no).	no
-r	--simulate no|initial|-random	Suppress simulation (no). Plan only for a specific activity (e.g. operational situation) of the activity automaton (initial). Run multi-step simulation of a sequence of activities by randomly resolving non-determinism in the activity automaton (random).	initial
-s	--statistics	Show information about risk structure generated from the model in FILE.	false
	--synthesise no|all|-controller	Suppress synthesis (no). Synthesise the controller (controller).	no
	--taxonomy	Generate LaTeX taxonomy of endangerments and mitigations, cf. Figures 3.2 to 3.4 and Table A.1.	false
	--version	Show version information.	false

2.5 Editing Yap Models in Emacs with yap-mode

Currently, there is a rudimentary Emacs major mode `yapp.el` included

- in the root directory of the ZIP package or
- in `/usr/share/yapp/` after installing the DEB package.

Add the following two lines to your `.emacs` to automatically enable that mode for `yap` files in your Emacs environment:

```
(autoload 'yap-mode pkgpath "Major mode for YAP script files" t)
(add-to-list 'auto-mode-alist '("\\.yap\\'" . yap-mode))
```

Key Binding	Description
`C-h m`	Display PDF manual for help.
`C-x C-a a`	Generate activity graph from current buffer.
`C-x C-a b`	Build artefacts for current buffer.
`C-x C-a c`	Generate safety controller model.
`C-x C-a f`	Generate fault tree from current buffer.
`C-x C-a p`	Simulate a random path through the activity graph, generate a LaTeX table showing some statistics of the simulation run, and risk graphs for each of the steps shown in this run.
`C-x C-a r`	Generate risk structure from current buffer.
`C-x C-a s`	Adjust format and viewer settings.
`C-x C-d c`	Display minimal cut sequences.
`C-x C-d g`	Display existing graphs generated from current buffer with the specified PDF viewer (e.g. `evince`, `zathura`).
`C-x C-d l`	Display YAP log file associated with current buffer.
`C-x C-d m`	Display model associated with current buffer.
`C-x C-d p`	Convert DOT file to PDF and display PDF.
`C-x C-d s`	Display current format and viewer settings.

Table 2.3: Emacs key bindings available in `yap-mode`

where, for example,

```
pkgpath = "/usr/share/yapp/yapp.el" or
pkgpath = "~/.emacs.d/lisp/yapp.el".
```

`yap-mode` currently supports syntax highlighting for YAP scripts, auto-indentation from the standard `c-mode`, as well as several key bindings (cf. Table 2.3) for executing some of the more frequently used command-line calls of YAP.

By default, generated artefacts will be placed in the sub-folder `output` of the directory with the currently open `.yap` file. Global logging is by default activated (see Table 2.1) and, when switched off, places log files in the same directory as the `.yap` file in the current buffer.

`yap-mode` aims to increase interactivity between the user and YAP models during the modelling process. Once an appropriate model is found, YAP's synthesis artefacts are supposed to be used in the down-stream controller design and development process.

In `yapp.el`, you may adjust several parameters such as, for example, logging, the default PDF viewer via `yap-viewer` or the default output directory via `yap-outdir` with the help of a rudimentary step-through wizard in the Emacs' mini-buffer.[7]

[7] I haven't had a lot of time yet for implementing an easy to readjust Emacs mode. Also my knowledge of Lisp has its limits. So, if you happen to have technical suggestions for improvement, please, don't hesitate to help out with a corresponding Lisp snippet.

Chapter 3

Yap's Input: Activities, Factors, and the Control Loop

For an appropriate conduct of its analyses, YAP requires information about relevant activities (Section 3.1), the control loop (Section 3.2), and relevant risk factors (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
2
3  [ Activity [ activityId ] { <Body> } ]
4
5    // another single line comment
6  [ ControlLoop loopName [ for activityId ] { <Body> } ]
7
8  [ FactorModel [ for activityId ] { <Body> } ]
9    /* a multi-line
10      comment */
```

Note 3 *For describing the* ***model syntax****, we use* `<Identifier>` *to denote structured non-terminals,* `<A|B>` *to signify choice among* `A` *and* `B`*,* `[A]` *to say that* `A` *is optional, and* `[A]`* *that* `A` *can occur zero or more times. Occasionally, we specify syntax using* BACKUS-NAUR *form rules introduced with "*`::=`*".*

Note 4 ***Keywords*** *are* ***case-insensitive*** *for the parser but* ***identifiers are case-sensitive****. Keywords may not be used as identifiers. You can stick with*

all-lower- or all-upper-case to avoid any confusion. YAP script supports Bash- and C-style (multi-line) comments. The ***default identifier*** *for activity, control loop, and factor specifications is the name of the main* `.yap` *file provided to YAP.*

Below, we will first discuss the directives `Activity`, `ControlLoop`, and `FactorModel`. The `Settings` part will be discussed later in Section 4.2. Most of the examples shown in this manual focus on the domain of *highly automated and autonomous driving*. However, YAP should be equivalently useful for many application domains where techniques such as HazOp, LOPA, STPA, or BTA are practised.

Note 5 *Technically, a YAP file can be empty, only leading to a warning about missing factor specifications when being processed. However, it is up to the user to decide about the usefulness of an empty model.* ☹

3.1 The Activity Model

First, YAP has to be provided with an *activity model*,[1] an abstraction of the control loop representing the processes running (or performed) in this loop. YAP supports the composition of activities into an *activity automaton*. For the classification of parts of the controlled process, we will use the term *aspect*. The partitioning of processes into aspects or activities requires domain and expert knowledge. It is out of the scope of this manual to dive into the details of this step. However, activities can be decomposed according to the concurrent and sequential processes comprising the loop, for example, "supply power", "driving", "operate vehicle." Such processes often determine the *functionality* of a system (Broy 2005, 2010).

Currently, YAP supports *sequential composition* and *superposition* of activities (see Section 5.3 for further details). Parallel composition is currently not possible. The activity model is a state-labelled finite automaton representing activities from an overall view of the controlled process. This view can be taken by a *safety or mitigation controller* synthesised from a YAP model. In summary, an activity model has to be crafted such that the considered process is faithfully described by performing one activity at a time.

In the file `example.yap`, the directive

```
1 Activity [ activityId ] {
2     <Body>
3 }
```

declares an activity labelled with `activityId`. `Body` describes the context of this activity and can be empty. `Body` can use the directives

- `description` (or its short form `desc`) for adding an informal description of the activity,

[1] At your convenience, this can be *operational situations* (OSs) to be mastered, tasks to be performed, user-level system functions to be applied.

- `include` for
  - **superposition** of an activity with the present activity,
  - **multiple inheritance** to reuse model information from other YAP files taking the role of libraries,
- `successor` for **sequential composition** of another activity performed after the present activity,
- `initialState` for specifying the *risk state* from which the exploration of the risk structure (Section 4.1) and mitigation planning will take place. *default:* 0

```
Activity [ activityId ] {
  [ <desc|description> "text"; ]
  [ <include|successor> activityPath; ]*
  [ initialState (rfId:phase[, rfId:phase]*); ]
}
```

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

Currently, YAP will parse **one** activity per file. For each identifier `activityPath`, YAP expects a file `activityPath.yap` to exist in the current or corresponding directory. Accordingly, `activityPath` can be either an activityId or a path ending with an activityId without the suffix `.yap`.

Example 1 (Modelling the Controlled Process) The following script applies the directives `include` and `successor`.

```
Activity parkWithRemote {
  include driveAtLowSpeed;
    successor autoLeaveParkingLot;
  successor leaveParkingLot;
}

ControlLoop myRoboCar for parkWithRemote {}

FactorModel for parkWithRemote {}
```

Figure 3.1 visualises this YAP script fragment using a flavour of Systems Modelling Language (SysML, Friedenthal et al. 2014) state charts. We will discuss the other directives in the activity model below and in Section 4.2. □

3.2 The Control Model

In addition to the activity model discussed in the previous section, YAP can be provided with a more detailed and more technical model of the *process of*

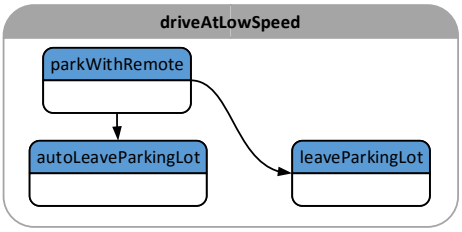


Figure 3.1: Activity fragment of the process declared in Example 1

interest to be controlled, also called the *application*, the *plant*, or the *control loop*. In `example.yap`, this can be declared by

```
<ControlLoop|Application> [ loopId [ for activityId ] ] {
    <Body>
}
```

For the easier modelling of complex systems, the primitives provided in YAP script support the engineer in maintaining a higher level of abstraction. For example, `Body` provides directives to specify behavioural *modes* of the control loop and physical *items* (or entities) *embodying* these modal behaviours.

```
ControlLoop [ loopId [ for activityId ] ] {
  [ mode modeId ... [ embodiedBy itemId ]; ]*
  [ [ item ] itemId ...; ]*
}
```

Details on how to specify *modes* are explained in Section 3.2.1 and the specification of *items* is described in Section 3.2.2.

Note 6 *In YAP script, we specify* attributes *in the form*

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

and relationships *in form of*

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

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

3.2.1 Mode Specifications

In YAP, three abstractions play together in a specific way. We have already discussed the first abstraction, the activity model in Section 3.1. YAP scripts contain risk models for the specified activities. Like activities, risk is modelled as

a kind of abstract state machine. The second abstraction comprises the notions of risk phases, factors, states, and spaces, capturing the state-based thinking of dependability engineers and risk analysts. The third abstraction extends these notions with abstract events and their refinements—operations, actions, or commands—, especially, of type nominal, endangerment, and mitigation. This abstraction is relevant for control engineers designing safety controllers for a particular machine.

Abstract state machines are action systems. Actions, more precisely, *guarded commands* are the units of behaviour we want to reason about. Such commands can themselves be complex programs. In YAP, actions are described in terms of *modes*. We use the term *mode* because a risk structure is not only in a particular activity and risk state but also in a particular mitigation mode.[2]

Modes specify mitigations, are useful for safety controller synthesis, and can be specified as part of the controller (Section 3.2) in the following way.

```
1  ControlLoop loopId {
2    ...
3    [ mode modeId [ alias modeName ]
4        [ <desc|description> "text" ]
5        [ role actionType ]
6        [ event eventName ]
7        [ <cause|guard> "embedded expression" ]
8        [ update "embedded expression" ]
9        [ target (param=val[, param=val]*) ]
10       [ embodiedBy itemId ]
11       [ param=val ]*; ]*
12   ...
13 }
```

`modeId` uniquely identifies the action, `modeName` can be used to provide an easier to understand identifier, and with `desc`, an informal description can be added. The `role` attribute refers to an `actionType` according to Table A.1. Currently, these classifiers informally characterise the modal behaviour and are used to generate transition labels in risk graphs output by YAP if more specific information is not available.

Controller synthesis requires that the controller to be synthesised is interacting with a wider system responsible for the nominal low-level or supervisory control of the machine. YAP currently supports PRISM's *probabilistic GCL* (pGCL, Kwiatkowska et al. 2007) following the *communicating sequential processes* (CSP, Hoare 1985; Roscoe 2010) approach to concurrent system design. That is, corresponding guard and update expressions can be embedded into a mode via the `guard` and `update` directives.

When specifying mitigations, `cause` and `update` can be used to specify what is often called a *safety function* responsible to remove the *cause* or causal factor

[2] I am inclined to change this terminology at some point but, mathematically, it does not matter which symbol we assign to this concept as long as we do it coherently.

from a currently active critical event or hazard. When specifying resumptions, `guard` and `update` can be used to specify the inversion (e.g. switching off) of this safety function if `guard` holds. Often `guard` will be the negation of `cause`.

The `event` directive can be used to specify synchronisations with machine modules external to the safety controller. All guarded commands with the same `eventName` are then executed synchronously.

If the wider system, the controller is embedded into, models the application in terms of *activities* (Section 3.1) and can be operated in several so-called *safety modes* (see, e.g. ISO/TS 15066 2016), then YAP provides the further directive `target`. `target` extends `update`, playing a special role in controller synthesis. The `param=val` pairs in `target` are used to synthesise the controller logic. With the above assumptions, one can specify by `act=a` a target *activity* a, the controller switches the machine to as part of a mitigation. Similarly, one can specify by `safmod=sm` a target *safety mode* sm, the controller switches the machine to as part of a mitigation. See the example in Gleirscher (2020).

Obviously, for these two parameters to work properly, the surrounding machine model must know about sm and a. For example, the machine must have an `act`-conjunct enabling certain commands in certain activities together with a logic switching between the activities according to the needs of the application. This switching is external to the safety controller but can be influenced by it. Analogously, sm must be implemented as a `safmod`-conjunct in the machine's commands, enabling and disabling subsets of commands in particular safety modes, with the difference that switching between safety modes falls under the sole responsibility of the safety controller. This concept has been elaborated and demonstrated in (Gleirscher and Calinescu 2020).

Further `param=val` pairs can be added to a mode specification to describe certain characteristics of the modal behaviour. YAP's synthesis facility currently supports integer parameters such as `disruption` and `effort`, useful to quantify the disruption of the nominal process and the effort to be spent when activating the corresponding mitigation mode. These parameters are then converted into reward structures used by tools such as PRISM for the search of optimal policies (Gleirscher and Calinescu 2020).

In summary, modes refine the STS with the risk state space and the three main classes of events (i.e., endangerments, mitigations, risk-neutral actions). Particularly, modes are a way to specify actions that refine these events into a composition of factor phase changes, activity or task changes, other changes of situational parameters.

Finally, the `embodiedBy` directive uses an `itemId` to indicate that a mode is implemented by a particular item. The item specifications required for this directive to work are described in the next section.

3.2.2 Item Specifications

In YAP script, items can be used to model the physical structure of the control loop.

```
ControlLoop loopId {
  ...
  [ [ <item|asset> ] itemId [ alias itemName ]
      [ <desc|description> "text" ]
      [ role itemType ]
      [ partOf itemId ]
      [ hasFunction modeId ]
      [ poweredBy (itemId[, itemId]*) ]
      [ param=val ]*; ]*
  ...
}
```

The following *attributes and relationships* can be used to specify items.

- `alias`: used to provide an easier to understand `itemName` for this item,
- `description`: attaches a free-text description,
- `role`: refers to an `itemType` according to Table A.2,
- `partOf`: specifies that this item is a part of another item,
- `hasFunction`: specifies that this item embodies the mode `modeId`, Experimental!
- `poweredBy`: specifies that this item requires an *energy source* that is provided by another item. Experimental!
- The `param=val` pairs can be used in a way similar to their use in mode specifications (Section 3.2.1).

The control loop identified by `loopId` is the top-level item and every item defined in this control loop is part of this loop. `asset` is a synonym for a particular form of `item` useful for impact modelling in risk assessments.

Example 2 (Control Loop Fragment for "supplyPower") The loop fragment for the aspect "supplyPower" consists of an item `Pwr` referring to the primary energy supply of the controller. `Pwr` is a physical part of the item `Ve`, the vehicle. `Bat` refers to a battery as the alternative energy source. Finally, an item `Ctr` is declared, powered by at least one out of the two energy sources `Pwr` and `Bat`.

```
ControlLoop supplyPower for supplyPower {
    mode pwrFailure embodiedBy Pwr;
    mode switchToBat embodiedBy Bat;

    Ve alias Vehicle;
    Pwr alias PrimaryEnergySource
        partOf Ve;
    Bat alias Battery
        partOf Ve;
```

```
10      Ctr poweredBy (Pwr, Bat);
11  }
```

3.3 The Factor Model

In YAP, one can specify faults, causal factors, hazards, incidents, accidents, or other mishaps and their relationships, usually known from HARA (Section 1.2). The *risk factor* is YAP's primitive used for this kind of risk modelling.

In our file `example.yap`, a *factor model*[3] is introduced through:

```
1  FactorModel [ for activityId ] {
2      <Body>
3  }
```

Note 7 *Although the specification of one factor model per YAP file is optional (cf. Chapter 3), YAP expects to have a non-empty factor model after having parsed all included YAP files.*

In Body, you can specify factors relevant for the activity activityId by using several directives.

```
1  FactorModel [ for activityId ] {
2    [ rfId [ alias factorName ]
3           [ <desc|description> "text" ]
4
5           [ direct ]
6           [ offRepair [(<rfId[, rfId]*>|*)] ]
7           [ <final|incident|accident|mishap> ]
8
9           [ <requires|requiresNot> (<rfId[, rfId]*|*>) ]
10          [ requiresNOf (lb '|' rfId[, rfId]*[ '|' ub ]) ]
11          [ <requiresMit|requiresOcc> (rfId[, rfId]*) ]
12          [ excludes (<rfId[, rfId]*|*>) ]
13          [ causes (rfId[, rfId]*) ]
14          [ permits (rfId[, rfId]*) ]
15          [ prevents (rfId[, rfId]*) ]
16          [ preventsMit (rfId[, rfId]*) ]
17          [ mitPreventsMit (rfId[, rfId]*) ]
18
19          [ guard "embedded␣expression" ]
20          [ <activatedBy|detectedBy> (actionSpec[, actionSpec]*)
                ]
21          [ mitigatedBy (actionSpec[, actionSpec]*) ]
```

[3] Also called "hazard model" in earlier versions of YAP and for downward compatibility.

```
22          [ resumedBy (actionSpec[, actionSpec]*) ]
23          [ alleviatedBy (actionSpec[, actionSpec]*) ]
24
25          [ impacts (assetId interval[, assetId interval]*)]
26          [ param=val ]*;
27    ]*
28  }
```

Similar to other modelling primitives in YAP script, `alias` specifies an easier to memorise *factor name* and `description` attaches a free-text description to the factor. Moreover, a factor specification can contain directives describing *relationships* with other factors useful for risk space exploration:

- `causes` specifies that the **activation** of a factor is **propagated** and activates other risk factors. Note that this constraint can be overridden by `prevents` constraints.
- `requires` specifies that the activation of a risk factor requires other factors to be **activated in advance** or simultaneously,
- `requiresNOf` specifies that the activation of a factor requires $n \in [lb..ub]$ out of a range of m specified factors to be **activated in advance** or simultaneously, with the lower bound `lb` $\in [0..ub]$ and the upper bound `ub`$\in [lb..m]$.
- `requiresNot` specifies that the activation of a risk factor requires other factors to be **deactivated in advance**,
- `requiresMit` specifies that the *activation* of a factor requires another factor to be **mitigated in advance**.
- `requiresOcc` specifies that the *activation* of a factor requires another factor to have occurred, that is, to be either **activated or mitigated in advance**. However, if that other factor has been deactivated again, this constraint will evaluate to `false`.
- `permits` specifies that the **activation** of any of the given factors is **explicitly permitted** if this factor is activated. Such permission acts as a weak form of the `causes` constraint or the dual of `requires`, and *overrides* any other constraint (such as `prevents`) that tries to inhibit the activation of the given factors. Hence, `permits` has to be used carefully when crafting a risk model.
- `excludes` specifies that the activation of a risk factor **superposes** or **invalidates** another risk factor.[4]
- `prevents` specifies that the **activation or mitigation** of this factor **prevents** (or inhibits) the **activation** of other factors. Note that this constraint can be overridden by `permits`.

[4] Unlike `prevents` constraints, `excludes` constraints do *not override* `causes` constraints.

- `preventsMit` specifies that the **activation or mitigation** of a factor **prevents** the **mitigation** of other factors.
- `mitPreventsMit` specifies that the **mitigation** of a factor **prevents** the **mitigation** of other factors.

In summary,

> `permits` overrides `prevents` **and** `prevents` overrides `causes`,

meaning that a factor f_1 that is permitted by another factor f_2 can occur even if its causation by a factor f_3 is prevented by a factor f_4.

Moreover, `requires` and `requiresMit` constraints are stronger than `requiresOcc`. Either of these two will by conjunction overrule `requiresOcc`. Also note that it is possible to specify inconsistent constraints, for example, f_2 `requires` f_1 conjoined with f_2 `requiresMit` f_1 will never allow f_2 to be activated.

Several directives fundamentally determine how a risk factor is interpreted when unfolded for risk space exploration:

- `direct` specifies that a factor can be mitigated in one step, without visiting an intermediate state[5].
- `offRepair` is the opposite of `direct` and specifies that a factor can only be mitigated by putting the **machine out of order**, for example, for the duration of a repair task. The optional list of identifiers (`rfIds`) specifies factors that will be deactivated (i.e., reset) when an `offRepair` mitigation is performed. Use the wildcard `*` to include *all non-inactive* (i.e., active or mitigated) factors. By default, an `offRepair` mitigation includes this factor and other factors only if they are also repairable off-line.
- Factors are by default mitigable and resumable (e.g. repairable). However, the switch `final` can be used to specify that a factor can only be activated, that is, the detection of the corresponding endangerment is the only event that matters, no actual/dedicated mitigations are performed. In other words, this switch declares a factor to be a Boolean variable that can only be switched to `true`, with one exception:

  Note that `final` blocks the mitigation of all factors it requires, for example, for a factor specification *`f1`* `requires(`*`f2`*`)final;`, in all states where f1 and f2 are active f1 will block the mitigation of f2.
- `mishap` and, for convenience, its two synonyms `incident` and `accident`, specify that this factor when activated terminates risk space exploration.

Further types of constraints will be included in future version of YAP. Furthermore, a factor specification can contain directives carrying details for controller design and synthesis:

[5] The intention behind `direct` is to model run-time behaviour such as fail-operational behaviour.

- `activatedBy` specifies that a factor is **activated** by a specific **action** (e.g. an endangerment) performed by a specific **item** (e.g. an uncontrollable agent in the machine's environment). The synonym `detectedBy` stresses that this action is recognised or detected as an endangerment, for example, by sensors of a safety monitor built into the machine. For that, the mode referred to is embodied by items of type `SENSOR` (Table A.2).
- `mitigatedBy` specifies that an activated factor is mitigated by specific **actions** performed by specific **items**.
- `resumedBy` specifies that a factor, after having been mitigated, allows the machine to perform specific **actions** to inactivate that factor and resume nominal operation.
- `alleviatedBy` specifies that a mishap that has been caused by that factor can be alleviated by performing specific **actions**.
- `guard` attaches an expression (e.g. a PRISM state predicate) with the *activation* condition for this factor, based on the state space of a lower-level model (e.g. a pGCL model for PRISM). `guard` is used as an auxiliary directive and does not require the specification of modes as opposed to the `*By` directives.

References to modes (i.e., actions) of the control loop (Section 3.2.1) are specified in the form `actionSpec ::= actionClass[.modeId]|.modeId`.

- `activatedBy` (and `detectedBy`) may refer to a single endangerment of class `actionClass` from Figure 3.3, which is identified by `modeId`.
- `mitigatedBy`, `resumedBy`, and `alleviatedBy` may refer to one or more mitigation, resumption, and alleviation options of class `actionClass` from Figure 3.4, each identified by `modeId`.

  Note 8 *Moreover, several factors can refer to the same mode. This gives rise to mitigation requirements for a mode. Any mode implementation referred to or embedded in the mode specification has to be capable of performing mitigations of all the factors associated with this mode. For example, if mode M is referred to from the two factors* F_1 *and* F_2 *then M is required to be applicable to any risk state with any subset of these two factors activated and perform the maximum possible mitigation, that is, to mitigate or (in case of* `direct`*) deactivate as many factors as possible.*

If `actionClass` is left out,[6] it defaults to `ENDANGER`, `MITIGATE`, `RESUME`, or `ALLEVIATE` respectively (Table A.1). If `modeId` is left out then the whole directive is semantically ignored but may be useful during modelling. The same modes can be used across different factor specifications.

Finally, a factor specification can contain directives for quantitative risk analysis:

- `impacts` specifies one or more negative *consequences*, each described in Experimental!

[6] Note that for correct parsing, the "`.`" currently needs to remain in front of `modeId`.

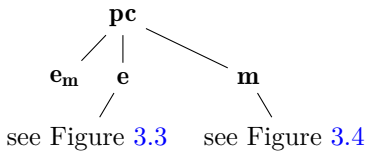


Figure 3.2: Taxonomy of actions; symbols are described in Table A.1 in Appendix A.1

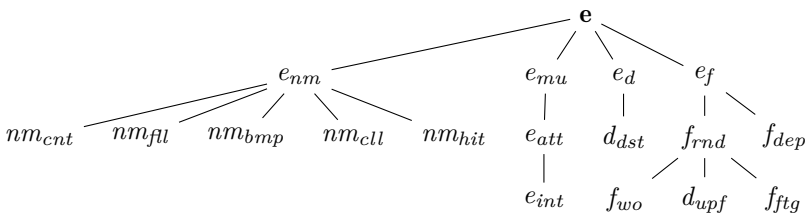


Figure 3.3: Taxonomy of endangerments; symbols are described in Table A.1 in Appendix A.1

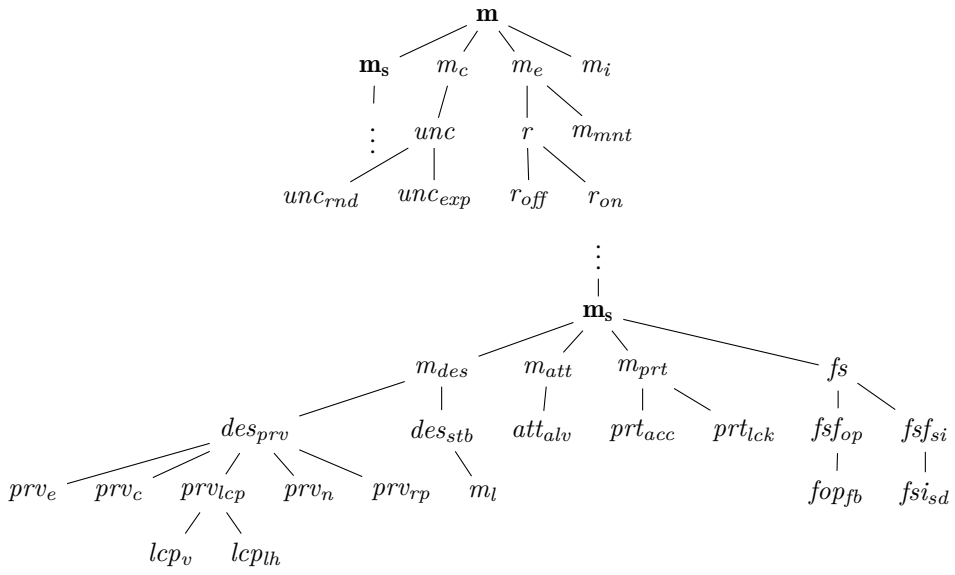


Figure 3.4: Taxonomy of mitigations; symbols are described in Table A.1 in Appendix A.1

terms of an **interval** (see Section 3.5 for interval specifications) quantifying the range of materialised impacts on a specific **asset** `assetId` when this factor is activated. To model the overall impact on a collection of assets, they can be composed into a single *compound asset* using `partOf` directives (Section 3.2.2).

- Factor specifications allow, like other YAP script primitives, the addition of *parameter-value pairs* according to Section 3.5.

  YAP' synthesis facility currently makes use of `prob` and `sev`, two parameters translated into a stochastic model for quantitative risk analysis. For example, `prob=0.05` states that there is a 5% chance of a mishap occurring from the corresponding factor being either unrecognised (i.e., a transition from its inactive phase to its mishap phase) or active and not mitigated (i.e., a transition from its activated phase to its mishap phase). Moreover, `sev=5` states that a mishap creates a general impact on not specified entities in the height of 5 "units of negativity." This way of using YAP is illustrated in more detail in Gleirscher and Calinescu (2020).

Below, we will see a number of examples that show how these directives can be used in factor specifications.

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

- `lowOrNo-Fuel` (F) describes states where a vehicle is soon running out of fuel or has already done so. The directive `offRepair` abstracts from the necessity of taking or pulling the vehicle to a petrol station. Here, the use of this directive suggests that the refill must take place while the vehicle is out of order.

- `lowOrNo-Energy` (E) describes states where, the vehicle controller is not supplied with sufficient electric power. `activatedBy (FAIL.pwrFailure)` specifies an endangerment of class `FAIL` (i.e., a failure) observed as the event `pwrFailure`, which activates E, and embodied by the item `Pwr`. `mitigatedBy (FALLBACK.switchToBat)` provide a mitigation of class `FALLBACK` performed by `switchToBat` and embodied by `Bat`. Finally, `offRepair` describes that this factor can only be completely reset after putting the vehicle out of operation.

- `lowOrNo-Battery` (B) specifies states where battery-based energy supply has ceased to work. `requires (E)` states that B only occurs in case of failure of the primary energy supply, that is, E. This abstraction includes two assumptions: the battery is continuously charged by `Pwr` and we neglect other causes of B, such as a broken battery or wire. □

Example 3 indicates how HARA techniques such as HazOp and FTA deliver information to be coded into YAP models: The `requires` constraint is an example

Listing 3.1: YAP script for the aspect "supplyPower"

```
Settings {
  suppressMishaps=false;
}

Activity {}

ControlLoop supplyPower {
  mode pwrFailure embodiedBy Pwr;
  mode switchToBat embodiedBy Bat;

  Ve alias Vehicle;
    Pwr alias PrimaryEnergySource
        partOf Ve;
    Bat alias Battery
        partOf Ve;
    Ctr poweredBy (Pwr,Bat);
}

FactorModel {
    F   alias lowOrNo_Fuel
        offRepair;
    E alias lowOrNo_Energy
    activatedBy (FAIL.pwrFailure)
        mitigatedBy (FALLBACK.switchToBat)
        offRepair;
  B alias lowOrNo_Battery
        requires (E);
}
```

of a piece of information possibly included from an FTA of the vehicle energy supply system. Furthermore, the factors F, E, and B share the common prefix "lowOrNo", showing an example of how one can apply HazOp guide-words, such as "too low" and "no", to the item `Ctr` of the vehicle's controller fragment `supplyPower`.

3.4 Impact Model

For quantitative risk assessment, YAP supports the specification of *impacts* from the activation of risk factors. Impact specifications are like constraints optional fragments of factor specifications and take the following form:

Experimental!

```
  FactorModel { ...
```

```
    rfId ...
        [ impacts (itemId [interval][, itemId [interval]]*) ]
        [ prob = Numeric ];
  }
  ControlLoop { ...
    asset itemId ...;
  }
```

With the keyword `impacts`, one can associate an arbitrary number of control loop items (Section 3.2.2) defined as `asset`s (i.e., humans, animals, environments, valuables of any kind) potentially negatively impacted after an activation of the factor `rfId`. The *severity* of the consequences from this activation, that is, the actual impact, is provided in terms of an interval (Section 3.5) that denotes the range of the best and worst expected impact. The *likelihood* can be given by a *probability* `prob` (numeric, $\in$ [0..1], Section 3.5). Risk r is then often calculated as a combination (in simpler cases merely a multiplication): $r = \texttt{prob} \cdot \texttt{sev}$, resulting in a *risk interval*.

Consider the following brief example of a chemical process with a valve X that when burst would cause poisonous bulk material to flow into a safe area:

```
FactorModel {
  VX desc "valve X bursts"
    mishap
    impacts (BM [5.2..10], A [8..17.3], O [10..50]);
}
ControlLoop {
  asset A desc "clean and safe area";
  asset BM desc "poisonous bulk material";
  asset O desc "health impairments of operators";
}
```

Here, we consider a threefold impact: the loss of bulk material valued from 5.2 to 10 negativity (or loss) units, the pollution of the clean area leading to cleansing costs in the height of 8 to 17.3, and health impairments of operators (10 to 50 injury units).

3.5 Parameter-Value Pairs

For activities (Section 3.1), control loops (Section 3.2), modes (Section 3.2.1), items (Section 3.2.2), and factors (Section 3.3), one can specify a collection of parameter-value pairs of the form `ParamId '=' Value` where

- `ParamId` is a unique alphanumeric identifier starting with a letter,
- `Value ::= Numeric | String | Interval`,
- `Numeric ::= [0-9]+['.'[0-9]+]?`,

- `String ::= '"' Text '"'` where `Text` is free-text not using the special characters `()[]{};,"'`, and
- `Interval ::= '[' Numeric '..' Numeric ']'` .

Note 9 *Although, for convenience, YAP script allows one to omit a leading 0 in floating point values (e.g. you can write* `x = .1` *for* `x = 0.1`*) and to omit quotes for strings without white-space (e.g. you can write* `s = thisIsAString` *for* `s = "thisIsAString"`*), such omissions are not encouraged. I will consider cancelling the support of this feature.*

Chapter 4

Yap's Output: Risk Structures

After having modelled factors for a specific control loop in a specific activity (see, e.g. Listing 3.1), we can use YAP to explore and output what we call a *risk structure* (Gleirscher, Calinescu, and Woodcock 2021). Again, we assume to have followed the advice in Section 2.2 and all example files are in the directory `~/yapp-examples/`. Then, we run YAP for the activity supplyPower using the command

```
yapp -m autodrv/supplyPower.yap \
     -o output/supplyPower.dot \
     -f latex -v 2 --simulate initial --nosimplify
```

and get a *risk graph* as an output. On the command line in your terminal, you should see something like

```
Memory usage: 10139920 bytes / 9 MiB
YAP's processing took 0.02 sec.
YAP processing information logged in autodrv/supplyPower.yap.log.
```

and the corresponding log file will contain something like

```
2020-09-11 08:56 CONFIG | Finished parsing command-line
  arguments.
2020-09-11 08:56 INFO   | Arguments:
  -m autodrv/supplyPower.yap -o output/supplyPower.dot -f latex
  -v 1 --simulate initial
2020-09-11 08:56 CONFIG | Starting YAP 0.6+20200911 ...
2020-09-11 08:56 INFO   | Identified control loop "supplyPower"
  for activity "supplyPower".
2020-09-11 08:56 CONFIG | No individual settings found. Applying
  default settings, see manual.
```

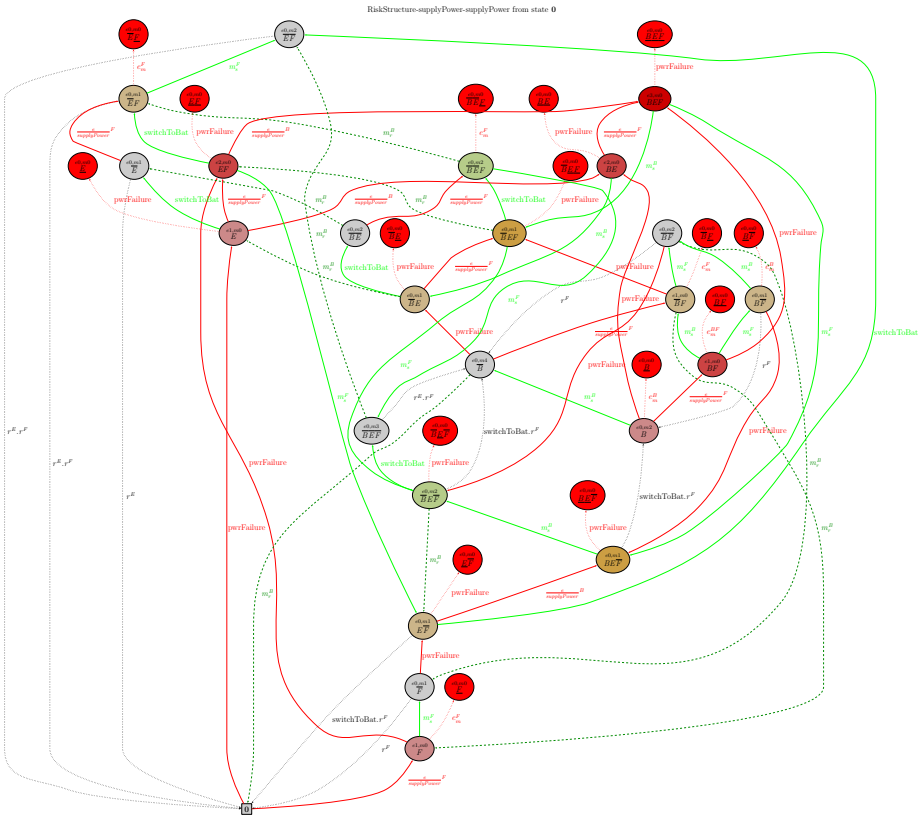


Figure 4.1: Risk graph generated by YAP from Listing 3.1 and representing the risk structure for supplyPower

```
2020-09-11 08:56 CONFIG | Finished parsing model file(s).
2020-09-11 08:56 CONFIG | Log level now set to SEVERE.
```

To transform this graph into a PDF file, we can use the commands

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

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

After using these commands, we should get a risk structure compliant with the graph shown in Figure 4.1. The file `supplyPower.yap` is included in the demonstration package, see Section 2.2.

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

Table 4.1: Phases of the risk factor `f` and their labelling

4.1 Understanding Risk Structures

Risk graphs visualise risk structures. It is time to establish a brief understanding of the semantics of risk graphs, such as the one in Figure 4.1. As shown there, a **risk structure** can be represented by a labelled directed graph. Risk structures are built by composing the **phase models** of the risk factors (Section 3.3) identified and relevant to a specific activity (Section 3.1). More elaborated phase models have been used. The basic phase model of a factor `f` consists of the four phases shown in Table 4.1.

Preliminary results of the algebraic theory underlying the construction of risk structures are published in Gleirscher (2014, 2017) and Gleirscher and Kugele (2017a,b) and further developed in Gleirscher, Calinescu, and Woodcock (2021).

Nodes. We distinguish several kinds of nodes in a risk graph, in other words, we consider several kinds of *risk states* in a risk structure:

- **0**: the (locally) "safest state" with none of the factors being activated. For the sake of brevity, given $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are not inactive, we simplify every label "$X0^{f_i}\,Y0^{f_j}\,Z$" to a label "XYZ" or, if empty, to "**0**".
- *Undesired events*: labelled with factors in either their active or mitigated phases. In Figure 4.1, the superscript eN in each node signifies the number of combined endangerments that have led to this state, mN denotes the number mitigations applied so far, where $N \in \mathbb{N}_0$.
- *Mishaps*: states that represent an unacceptable event (e.g. an incident or accident), indicated in red. The factor, say `f`, that is supposed to be the most influencing factor is switched to its mishap phase $\underline{f}$.

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

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

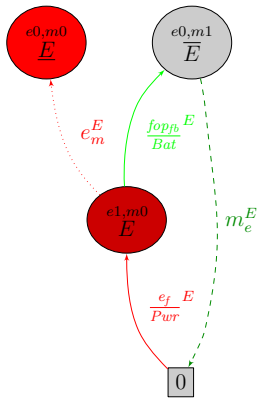


Figure 4.2: Phase model instantiated for the risk factor E from Example 3

Table A.1 in Appendix A.1 refines these categories into an action taxonomy. Furthermore, the edge labels in the graph correspond to the symbols given in the **Symbols** column of this table unless the labels can be derived from modes referred to from a factor specification (Section 3.2.1).

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

Embodiment. We have already made use of a further modelling element in Example 3: the concept of *embodiment* of an action, particularly, the *implementation* of an endangerment or mitigation. For example, the transition labelled with $\frac{e_f}{\texttt{Pwr}}^{E}$ specifies that the failure e_f activating the factor `E` stems from (electro-physical) behaviour embodied by the item `Pwr`. Moreover, the transition labelled with $\frac{fop_{fb}}{\texttt{Bat}}^{E}$ specifies that the fail-operational behaviour (*fop*) is realised 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>
}
```

can be used to adjust a range of settings that YAP takes into account when performing simulations, exploring and reasoning about risk structures, and synthesising controllers. In `Body`, you can use directives according to the syntax specified in the following listing.

```
Settings {
```

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

    [ <suppressMishaps
      | suppressEndangerments
      | suppressMitigations
      | suppressResumptions> = <true|false>; ]*

    [ <suppressMishaps
      | suppressEndangerments
      | suppressMitigations
      | suppressResumptions> = <true|false>; ]*

    [ allFactorsDirect = <true|false>; ]

    [ <suppressStateLabel
      | suppressMishapLabel
      | suppressEndangermentsLabel
      | suppressMitigationsLabel> = <true|false>; ]*

    [ grayscale = <true|false>; ]
    [ graphDirection = direction; ]
    [ extFile = path; ]
    [ requiredVersion = "prefix"; ]
}
```

`natNumber` must be a non-negative integer, that is, a number in $\mathbb{N}_0$. YAP generates risk structures always from a given *initial risk state*.

- `allFactorsDirect`, if `true`, declares all factors registered in the model to be `direct` (Section 3.3). *default:* `false`
- `endangermentDepth [0..MAX]`[1] specifies the maximal number of factors activated (whether sequentially or simultaneously) in a risk state at the same time. 0 specifies the exploration of the full endangerment space. *default:* 0 (maximum)
- `mitigationDepth [0..MAX]` specifies the total number of mitigations taken into account (in sequence or simultaneously) when planning a mitigation strategy. 0 specifies the exploration of the full mitigation space. *default:* 0 (maximum)
- `simulationLength [0..MAX]` specifies the number of steps to be executed out of a *scenario* or *run* of a controlled process (Section 3.1), 0 and 1 are equivalent to `--simulate initial`. *default:* 5
- `outputDepth [0..MAX]` specifies the depth of the exported risk graph *default:* 0 (maximum)

[1] `MAX` is currently equivalent to the JDK constant `Short.MAX_VALUE = 32767`.

measured as the path length from the `initialState` (by default state **0**, see Section 3.1). 0 specifies the export of the full graph.

default: 1.0
- `riskDiscount [0..1]` specifies the discount parameter for calculations of the discounted expected risk (Sections 3.4 and 5.5).

Furthermore, to control the parts of a risk structure to be included in the output, YAP provides the following switches:

default: `false`
- `suppressEndangerments`, if `true`, suppresses all endangerment transitions in the risk graph.

default: `false`
- `suppressMitigations`, if `true`, suppresses all mitigation transitions in the graph.

default: `true`
- `suppressMishaps`, if `true`, suppresses the annotation of the risk graph with mishap states.

default: `false`
- `suppressResumptions`, if `true`, suppresses all transitions leading to phase 0^f of a risk factor `f`.

The combination of `suppressMitigations` and `suppressResumptions` ensures that a risk structure is composed of loop-free phase models (i.e., acyclic factor semantics), resulting in an acyclic risk structure and risk graph.

default: `false`
Switches of the kind `suppress*Label` are useful to control details of the graphical output, for example, labels of specific classes of states and edges.

default: `BT`
The setting `graphDirection` allows the adjustment of the GraphViz `dot` setting for the layout `direction` of a graph (e.g. `BT`, `TB`, `LR`, `RL` where `B`, `T`, `R`, `L` stand for bottom, top, right, and left).

default: `false`
With the setting `grayscale`, YAP only uses grayscale for risk graphs.

default: version of the used YAP binary
The setting `extFile` can be used to specify the *path* to an external template file for controller synthesis, and `requiredVersion` allows one to indicate that the model at hand is to be processed by specific versions of YAP. Particularly, *prefix* is matched with the version string of the installed YAP binary.

Example 4 (Reducing Risk Graphs, Yap's Output) Given the file `supplyPower.yap`, for example, setting `suppressEndangerments` to `false` in risk state **0** would lead to an empty risk structure as there is no risk state a mitigation can be applied to, see Figure 4.3a. As an alternative to Figure 4.1, we chose to omit mishap states by using `suppressMishaps = true` leading to Figure 4.3b. Furthermore, with `suppressResumptions = true` we obtain Figure 4.3c. Finally, `suppressMitigations = true` results in a risk graph reduced to the set of *undesired events* directly reachable by all possible combinations of endangerments or factor activations respectively, see Figure 4.3d. □

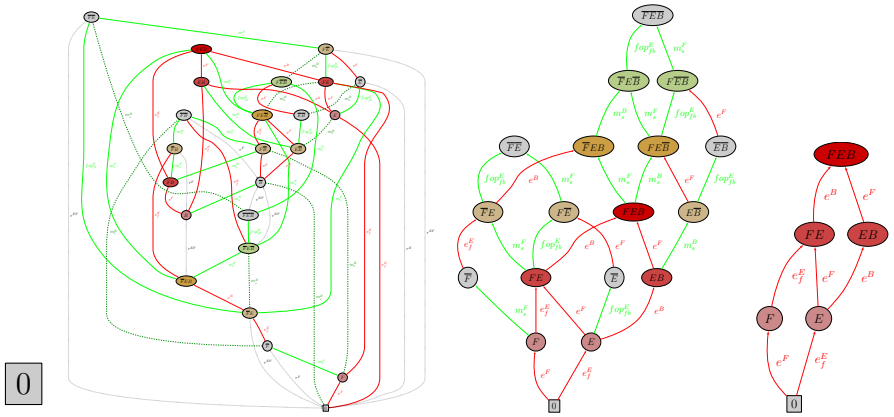


Figure 4.3: Suppressing parts of a risk graph

Chapter 5

Working with Yap

Yap's output can be used to perform two important steps in 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).

Additionally, this section discusses several aspects relevant when working with Yap, for example, property inheritance (Section 5.3), using wildcards (Section 5.4), inspecting risk spaces, and finally, mitigation orders (Section 5.5). The generation of safety controllers as conceptualised in Gleirscher and Calinescu (2020) is explained in Gleirscher (2020).

5.1 Reduction and Shaping of Risk Structures

As introduced in Section 3.3, dependencies between risk factors can be specified in terms of constraints, such as `causes`, `prevents`, `requires`, or `excludes`. Many examples in this manual make use of constraints and provide the corresponding rationale for their use, for example, Example 3 or Listing 3.1.

It is beyond the scope of this manual to describe methodological aspects of why, when, and where to use these directives. However, these directives support the shaping and simplification of risk structures for run-time mitigation planning. A more fundamental treatment of constraints is presented in the work of Gleirscher, Calinescu, and Woodcock (2021).

5.2 Performing Symbolic Simulation

Let us assume that we have created a bunch of Yap scripts for a controlled process, each script file modelling an activity, and activities superimposed using `include` directives and connected using `successor` directives (Section 3.1).

Consider the `autodrv` case study, which is an example of such complexity. There, we can type

```
yapp -m autodrv/start.yap \
     -o output/start.dot \
     -f latex -v 0 -l 1 --simulate random -s
```

to run a random simulation starting from the activity `start`.

Then, YAP step by step picks a random successor until the `simulationLength` (Section 4.2) 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 activity described in the file `start.yap`.

Experimental!

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

```
1  Settings {
2    outputDepth = 2;
3    endangermentDepth = 1;
4    mitigationDepth = 1;
5    simulationLength = 20;
6    suppressMishaps = true;
7    suppressEndangerments = false;
8    suppressMitigations = false;
9    suppressResumptions = true;
10 }
```

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 10 factors. This run took about 0.5 sec. □

YAP's algorithm for risk space exploration performs a breadth-first search through the reachable risk space. Here, reachability is defined by the constraints used in a YAP model. This procedure corresponds to the construction of reachability graphs in explicit-state model checking, with the main difference that YAP's constraint mechanism only covers a small fragment of the expressions allowed in guards of propositional GCL. Moreover, YAP does not carry through any property checking. Obviously, YAP's algorithm is exposed to the *state space explosion problem* of explicit-state model checking (Baier et al. 2008). Section 6.2 highlights some implications of that problem for YAP.

5.3 Property Inheritance and Superposition of Sub-Models

This section discusses a few aspects to deal with when crafting larger models encompassing several YAP files.

Table 5.1: Simulation run showing 20 steps. Values in parentheses indicate that, for `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

The `include` directive allows for a *superposition* of activity and factor models (Sections 3.2.1 and 3.3). Moreover, this directive enables the *inheritance of properties* (e.g. factor specifications) from included YAP files. YAP supports multiple inheritance. Because YAP files can be recursively included, large models can contain an inheritance hierarchy.

Inheritance through superimposition works on both activity and factor models. For example, if activity A `include`s another activity B then A inherits all model fragments of B, particularly, the activity model with its `successor` directives, the factor model, the control loop model, and potential settings. Importantly, only A will remain in the activity model as a *base activity*. Notice that the inheritance of B by A will not have any impact on activities connected from A via `successor` directives.

Moreover, a factor specification, say of factor f, in activity B will become a factor specification of A. If f already exists in A, the fragment of f in A will be *merged and overwritten* by the fragment of f in B. The crafting large models with these mechanisms may benefit from `include` as an abstraction facility. However, the use of this facility still implies care for maintaining model validity.

YAP collects settings *globally* via the `Settings` directive (Section 4.2). Differently from the factor models, which are specific to a basic activity constructed by `include` only, settings and control loop fragments are collected in a single place when recursively traversing the `include` and `successor` directives.

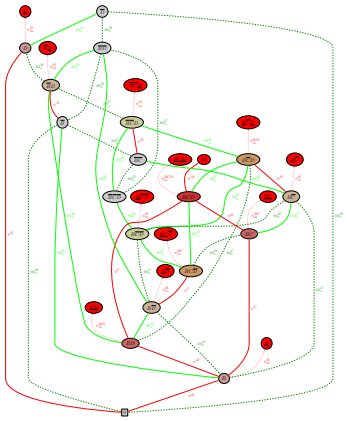
5.4 Using Wildcards

The use of wildcards is in the same way convenient and intricate. This is so because of an anyway relatively high level of abstraction risk structures are supposed to be used for. Wildcards can be seen as a way of dynamic constraint typing, hence, making constraints more concise and more powerful on the one hand but leading to a less obvious semantics on the other. Currently, wildcards can be used with the following constraints:

- `requires`: The expression `f requires (*)` signifies that the activation of factor f requires the activation of all other factors specified for this activity except for f itself.

```
// Using wildcards
Activity wildcard {}

FactorModel for wildcard {
  M requires (*)
    mishap;
  B; // explicit default
       specification
  C requires (B);
  D prevents (B);
}
```



- `excludes`: The expression `f excludes (*)` signifies that factor f, once activated in a particular risk state, cuts any further analysis from that state.

- `offRepair`: The expression `f offRepair (*)` states that factor f needs to be mitigated by putting the machine out of order and that the mitigation of f fully mitigates all other factors as well.

5.5 Enumerating and Ordering Risk Spaces

The 40 nodes in Figure 4.1 are risk states of the risk structure explored for the activity supplyPower. The states can be ordered by decreasing risk or severity level using a concept called *mitigation order* (Gleirscher and Kugele 2017b). Currently, YAP calculates a refined version of the *strong mitigation order* (Gleirscher, Calinescu, and Woodcock 2021). This is a linear order over risk states based on *severity intervals* over floating-point numbers provided for each factor as shown in Listing 5.1.

Listing 5.1: Activity supplyPower from Listing 3.1 extended by severity intervals

```
FactorModel {
```

```
    F  alias lowOrNo_Fuel
       offRepair
    sev=[5..10];
    E alias lowOrNo_Energy
    activatedBy (FAIL.pwrFailure)
       mitigatedBy (FALLBACK.switchToBat)
       offRepair
    sev=[1..6];
  B alias lowOrNo_Battery
       requires (E)
    sev=[2..7];
}
```

With the command

```
yapp -m autodrv/supplyPower-sev.yap \
     -o output/supplyPower-sev.dot \
     -f latex -v 1 --simulate initial --nosimplify --severity
```

YAP enumerates the 40 states explored and ordered by descending *risk/severity level or priority*. The result for the example in Listing 5.1 is stored in a file suffixed with `_riskspace.txt`, for example, `supplyPower_riskspace.txt`.

No.	State	Severity	14	BEF	[1.0..10.0]	28	$\overline{B}E$	[1.0..6.0]
1	$\overline{B}\underline{E}\underline{F}$	[5.0..10.0]	15	$\underline{E}\underline{F}$	[1.0..10.0]	29	$\underline{E}\overline{F}$	[1.0..6.0]
2	$\overline{B}E\underline{F}$	[5.0..10.0]	16	$\overline{B}\underline{E}\underline{F}$	[1.0..10.0]	30	E	[1.0..6.0]
3	$\overline{E}\underline{F}$	[5.0..10.0]	17	B	[2.0..7.0]	31	$E\overline{F}$	[1.0..6.0]
4	$\overline{B}F$	[5.0..10.0]	18	$B\overline{F}$	[2.0..7.0]	32	$\underline{E}$	[1.0..6.0]
5	F	[5.0..10.0]	19	$\underline{B}$	[2.0..7.0]	33	$\overline{F}$	[0.0..0.0]
6	$\overline{B}\underline{F}$	[5.0..10.0]	20	$\underline{B}\overline{F}$	[2.0..7.0]	34	$\overline{E}$	[0.0..0.0]
7	$\overline{E}F$	[5.0..10.0]	21	$\underline{B}\underline{E}\overline{F}$	[1.0..7.0]	35	**0**	[0.0..0.0]
8	$\underline{F}$	[5.0..10.0]	22	$B\underline{E}$	[1.0..7.0]	36	$\overline{B}\overline{E}\overline{F}$	[0.0..0.0]
9	BF	[2.0..10.0]	23	$B\underline{E}\overline{F}$	[1.0..7.0]	37	$\overline{B}\overline{E}$	[0.0..0.0]
10	$\underline{B}\underline{F}$	[2.0..10.0]	24	$\underline{B}\underline{E}$	[1.0..7.0]	38	$\overline{B}$	[0.0..0.0]
11	$\overline{B}EF$	[1.0..10.0]	25	$\overline{B}\underline{E}\underline{F}$	[1.0..6.0]	39	$\overline{B}\overline{F}$	[0.0..0.0]
12	$\underline{B}\underline{E}\underline{F}$	[1.0..10.0]	26	$\overline{B}E\overline{F}$	[1.0..6.0]	40	$\overline{E}\overline{F}$	[0.0..0.0]
13	EF	[1.0..10.0]	27	$\overline{B}\underline{E}$	[1.0..6.0]			

5.6 Displaying Activity Graphs

For models with an elaborate activity structure based on the `successor` and `include` relation, YAP allows one to generate an activity graph such as the one in Figure 5.1. Such graphs highlight the structure of the basic activities (blue nodes) created by the `successor` relation (solid edges) as well as the factor

model inherited from more general activities (gray nodes) via the `include` relation (dashed gray edges). By using the CLI switch `-v 1`, you can add legend labels to the activity graph for better readability.

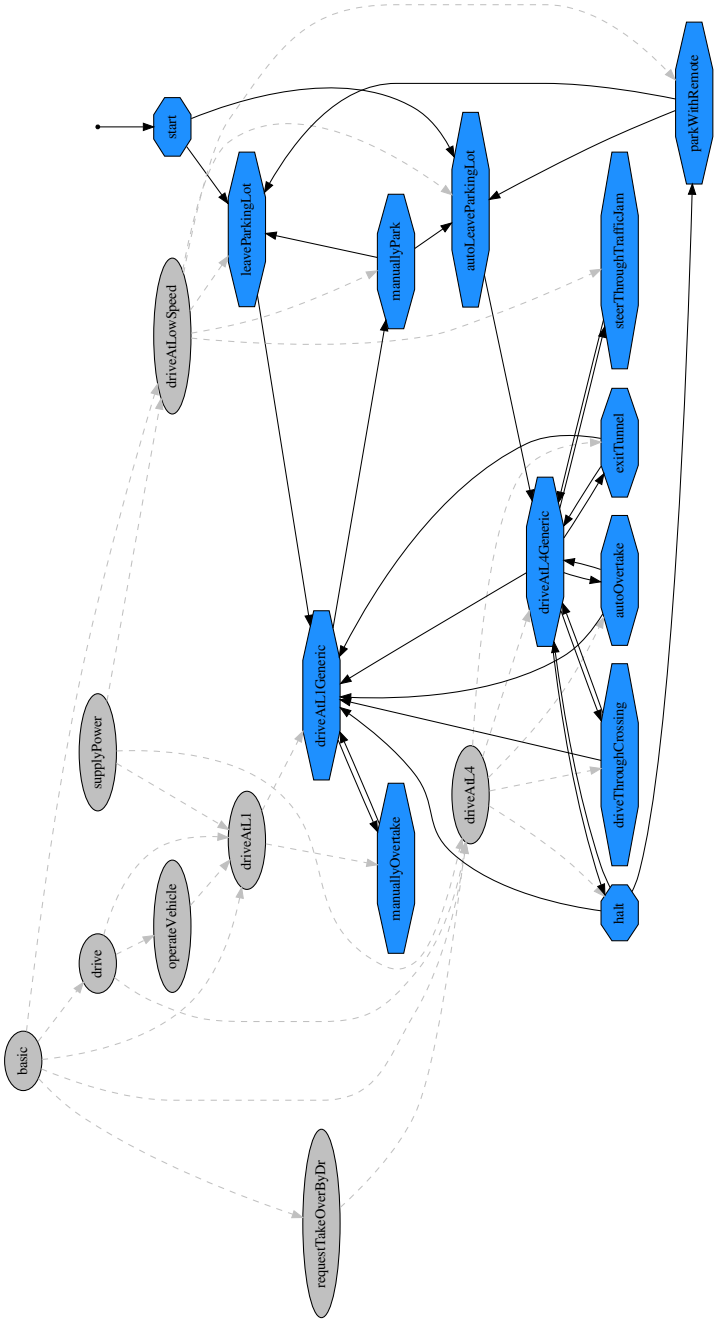


Figure 5.1: Activity graph generated from `start.yap` in Section 5.2

Chapter 6

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.

6.1 Frequently Asked Questions and Troubleshooting

Frequently Asked Questions. None reported or relevant so far.

Model Debugging and Further 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 information about its internal processing (e.g. warnings, error messages, notes about processing steps) to a log file:

- By default, this log file is associated with the `FILE` specified via the CLI option `--model`, see Section 2.4. Accordingly, you will find a file called `FILE.log` in the directory with the corresponding YAP script.
- If the CLI option `--global-logging` is provided, the log file is stored in the user's home directory, in Linux under `~/.yap/yap.log`.

Furthermore, you can raise the log level using the option `--log` to get more detailed information.

6.2 Known Limitations and Bugs

The following list highlights some technical and methodological limitations.

1. YAP is a research tool prototype with *demonstration and proof-of-concept* as its preliminary design goals and, thus, exhibits a number of algorithmic complexity issues that can be reduced, and a number of optimisations in time and memory consumption that have not yet been applied. Fixes to part of these issues are known and on my agenda.

2. For controller synthesis, YAP collects fragmented factor specifications in one place. Fragments in neighbouring activity specifications can overwrite each other's *shared factor attributes*, which is not usually desirable. The reason for this behaviour are some constraints in the way model data is currently handled internally.

   The easiest fix is to *keep factor specifications in one place*, preferably, in the most general included activity file.

   However, for activity graph traversal and risk space exploration, fragmented factor specifications will be composed *locally per activity* via the inclusion/inheritance hierarchy. Hence, overwriting occurs only, and as usually desired, downwards the inclusion hierarchy, that is, *factor fragments in sub-activities overwrite factor fragments in super-activities*.

3. YAP does neither parse external target models syntactically nor semantically. Hence, the procedure explained in Gleirscher (2020) is coming along with specific expectations on how the target model needs to be structured (e.g. using template parameters in appropriate locations) and requires the engineer to encode part of the "glueing information" into the YAP model. This is certainly sub-optimal but hopefully acceptable for a proof-of-concept tool freely available.

Bugs. No relevant unfixed bugs known so far.

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 Actions

Table A.1 provides the currently supported list of *action types*. Please, refer to Section 3.3 or Section 3.2.1 for the usage of action types.

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

actionType	Symbol	Description of the Action Type
PHASECHANGE	pc	Class encompassing all action classes.
SKIP		No operation. In CSP, termination.
UNDEFINED		The undefined action.
ENDANGER	e	Class of all endangerment actions or critical events.
MISHAP	e_m	Mishap actions leading to risk states (i.e., undesired events such as accidents or incidents) from where there is (currently) no feasible or acceptable 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.
SHARE	e_{sh}	Human-machine or human-robot collaboration taking place in physically shared space. Critical share events might also result from overlapping trajectories forming on-collision-course situations.
NEAR_MISHAP	e_{nm}	Actions exhibiting reducible events, e.g. collisions which can be alleviated, near-collisions on the road.
ATTACK	e_{att}	Attack actions, e.g. IT attack, soft or physical attack.
INTRUDE	e_{int}	Intrusion actions, e.g. traceable unauthorized access.

cont'd on next page

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

actionType	Symbol	Description of the Action Type
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.
FAULT	e_{f2}	Envisaged cause of failure (see e_f).
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.
MITIGATE	m	Class of all mitigation actions, i.e., for hazard treatment, intervention, pre-emption
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.
RESUME	m_r	Class of successful completions for partial mitigations.
COMPLETELY_MITIGATE	m_c	Class of mitigations directly returning from the active to the inactive phase.
FAIL_SAFE	fs	Class of mitigations dealing with defect treatment.
DE_ESCALATE	m_{des}	Actions preventing from the occurrence of (or de-escalating) a hazardous event or situation.
PROTECT	m_{prt}	Actions protecting valuable assets by access restriction, 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	m_{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.

cont'd on next page

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

actionType	Symbol	Description of the Action Type
INTERLOCK	prt_{lck}	Actions controlling physical access to shared resources like such as rail tracks, road crossings, flight route segments, collaborative work spaces by mechanisms e.g. road traffic lights, train interlocking systems, air traffic control.
ALLEVIATE	att_{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 or usage of physical actuators, e.g. ABS.
EVACUATE	prv_e	Actions performing evacuation of a dangerous area.
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_{lcp}	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	lcp_v	Mechanisms for vigilance checking, e.g. dead man switch, driver fatigue detection.
LIMP_HOME	lcp_{lh}	Mechanisms for short- or medium-term (minimum risk) navigation to a safe location.
H2M_HANDOVER	lcp_{ho}	Hand over to human operator when preconditions for controller usage cease to be met.
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.
RESTART	r_{rst}	Actions resetting or restarting parts of the controller, e.g. in case of soft errors.
FAIL_SILENT	fsf_{si}	Actions representing mechanisms for deactivating or 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. Restart actions are not taken into account here!
FALLBACK	fop_{fb}	Actions representing degradation to a backup component of the controller.

A.2 Taxonomy of Items

Table A.2 shows the currently supported list of *item types*. Please, refer to Section 3.2.2 for the usage of item types.

Table A.2: Comprehensive list of currently supported item types

Item Type	is a/is part of	Description of the Item Type
UNSPECIFIC		item role not specified
CONTROL_LOOP	UNSPECIFIC	the overall controlled process
ACTOR	CONTROL_LOOP	an active entity performing actions in the loop
AGENT	ACTOR	synonym for actor
CONTROLLER	AGENT	controller responsible for safety functions
PLANT	AGENT	a collection of machines, robots, and other installations
HUMAN_OPERATOR	CONTROL_LOOP	a human operator working in the plant
SENSOR	CONTROLLER	generic sensor for sensing everything
ACTUATOR	CONTROLLER	
COMPUTING_UNIT	CONTROLLER	
NETWORKING_UNIT	CONTROLLER	
HMI_CONTROL	SENSOR	
HMI_DISPLAY	ACTUATOR	
SOFTWARE_COMPONENT	COMPUTING_UNIT	
ENVIRONMENT	UNSPECIFIC	a passive entity of any kind
REGION	ENVIRONMENT	a spatial entity, a geometric area or volume, typically stationary
MOBILE_OBJECT	ENVIRONMENT	mobile objects of any kind
VEHICLE	MOBILE_OBJECT	vehicles of any kind
WORK_PIECES	ENVIRONMENT	countable manufacturing artefacts
MATERIAL	ENVIRONMENT	uncountable bulk material, fluids, or powders
OPTICAL	SENSOR	
RESISTIVE	SENSOR	
CAPACITIVE	SENSOR	
INDUCTIVE	SENSOR	
ACCUSTIC	SENSOR	
MECHANICAL	SENSOR	
THERMOELEC	SENSOR	
MAGNETIC	SENSOR	
PIEZOELEC	SENSOR	
LIGHTBARRIER	INDUCTIVE	light barrier sensing objects crossing
RANGEFINDER	INDUCTIVE	
LIDAR	SENSOR	
RADAR	SENSOR	
LASER	SENSOR	
SONAR	ACCUSTIC	
TACTILE	SENSOR	
TSBUTTON	TACTILE	touch-sensitive button
FSRESIST	TACTILE	force-sensing resistor or potentiometer
MICROPHONE	SENSOR	
CAMERA	OPTICAL	

A.3 Property Library

Table A.3 provides a comprehensive overview of YAP's library of properties, for example, *probabilistic computation tree logic* (PCTL) properties for MDP verification.

Table A.3: Comprehensive list of properties for controller verificaton

PropertyId: Property	Tags	Description of Property
DEADLOCK_FREEDOM_INITIAL: `A [ G !"deadlock" ]`	MODEL_TESTING	Can a deadlock be reached from the initial state?
EXPECTED_SEVERITY_UB: `R{"risk_sev"}<=s [ S ]`	NON_PARAMETRIC	Is the average expected severity in the long-run no greater than p?
HAZARD_POSS: `filter(exists, E [ F "RF" ], "init")`	SINGLE_EVENT	Can hazard RF occur in the model? (I.e. is it relevant? Show a shortest scenario where RF occurs.)
HAZARD_INEV: `A [ F "RF" ]`	SINGLE_EVENT	Is hazard RF inevitable?
HAZARD_PROB_UB: `P<p [ F "ANY" ]`	NON_PARAMETRIC	Is the probability of the occurrence of ANY hazard below p? (for MDPs: is the maximum probability of that path property across all adversaries below p?)
HAZARD_PROB_LB: `P>p [ F "ANY" ]`	NON_PARAMETRIC	Is the probability of the occurrence of ANY hazard beyond p?
HAZARD_INEVIT: `A [ F "RF" ]`	SINGLE_EVENT	Is the hazard inevitable? (Is the hazard so generic that the safety controller has to intervene by default?)

CTR_LIVELY:
`A [ F (RFp=act => (A [ F RFp=mit => (A [ F RFp=inact ])`
`])) ]`
	SINGLE_EVENT	Is the safety controller lively handling hazard RF in all situations?

MON_LIVELY: `A [ F ("RF" => (A [ X RFp=act ])) ]`	SINGLE_EVENT	Is the safety monitor on all paths immediately recognising the hazard RF?
HAZARD_IN_PROD: `E [ F ("RF" \& !"FINAL") ]`	SINGLE_EVENT	Can the hazard RF occur during a production cycle?
MISHAP_ANY_LOW: `S<p [ "MISHAP" ]`	SINGLE_EVENT	Is the steady-state (long-run) probability of any mishap RF below p?
DEADLOCK_FREEDOM_GLOBAL: `filter(forall, A [ G !"deadlock" ])`	MODEL_TESTING	Can a deadlock be reached from any state?
HAZARD_MON_FEAS: `E [ CE_RF => (X RFp=act) ]`	SINGLE_EVENT	Does the monitor immediately recognise RF in at least a single situation?
HAZARD_MON_ALL: `A [ F (CE_RF => (X RFp=act) ) ]`	SINGLE_EVENT	Does the monitor immediately recognise RF in all situations?

cont'd on next page

Table A.3: Comprehensive list of properties for controller verificaton (cont'd)

PropertyId: Property	**Tags**	**Description of Property**

SUCC_MIT:
`A [ G ( ("ANY" | RFp=act) => ( F<=t (RFp=mit \& !"ANY") ) )`
`]`
	SINGLE_EVENT	Does the controller always (i.e. along all paths starting from the initial state) mitigate hazard RF within t steps?

SUCC_RES: `A [ G ( RFp=mit => ( F<=t RFp=inact ) ) ]`	SINGLE_EVENT	Does the controller always (i.e. along all paths starting from the initial state) resume from mitigated hazard RF within t steps?
SUCC_CYCLE: `E [ RFp=act => (F RFp=mit => (F RFp=inact)) ]`	SINGLE_EVENT	Is the critical event handler for RF feasible (i.e. does there exist a successful run)?

TASK_ACHIEVEMENT:
`A [ G ( ( !"ANY" \& "pre_A" ) => ( ( F ( "post_A" ) ) W "ANY"`
`) ) ]`
	CTR_VERIFICATION	Does the plant achieve task A (post) given A is requested (pre)?

TASK_ACHIEVEMENT2: `A [ F G "FINAL" ]`	CO_SAFETY	Does the plant always reach and maintain the final state?
TASK_ACHIEVEMENT3: `A [ F "FINAL" ]`	REACHABILITY	Does the plant always reach the final state?
TASK_FIN_WITH_HAZ: `E [ F "RF" \& (F "FINAL") ]`	SINGLE_EVENT	Can we finish our task (at least once) after hazard RF has occurred (once)?
TASK_FINISH: `E [ F "FINAL" ]`	MODEL_TESTING	Can we finish our task (at least once)?
TASK_FINISH2: `E [ F "hFINAL_CUSTOM" ]`	MODEL_TESTING	Can we finish our task (at least once)?
TASK_FINISH3: `A [ "init" => (F ("hFINAL_CUSTOM" | "MISHAP")) ]`	MODEL_TESTING	Can we finish our task (at least once)?
TASK_FINISH4: `P>=pfin [ "init" => ((F "hFINAL_CUSTOM") \& (G !"MISHAP")) ]`	MODEL_TESTING	Can we finish our task (at least once)?
HAZARD_PROB: `Pmax=? [ F "RF" ]`	SINGLE_EVENT	What is the maximal probability of the critical event RF?
HAZARD_PROB2: `P=? [ "init" => (F "RF") ]`	SINGLE_EVENT	What is the probability of the critical event RF?
ANY_HAZARD_PROB: `P=? [ F "ANY" ]`	NON_PARAMETRIC	What is the overall probability of occurrence of ANY of the specified hazards?

cont'd on next page

Table A.3: Comprehensive list of properties for controller verificaton (cont'd)

PropertyId: Property	Tags	Description of Property
HAZARD_MAXPROB: Pmax=? [F "ANY"]	NON_PARAMETRIC	What is the maximum probability of occurrence of ANY of the specified hazards?
HAZARD_MINPROB: Pmin=? [F "ANY"]	NON_PARAMETRIC	What is the minimum probability of occurrence of ANY of the specified hazards?
HAZARD_PRNG: Pmax=? [F CE_RF] - Pmin=? [F CE_RF]	SINGLE_EVENT	What is the probability range of hazard RF?
MISHAP_PROB: P=? [F RFp=mis]	SINGLE_EVENT	What is the probability of a mishap from hazard RF?
MISHAP_ANY_PROB: P=? ["init" => (F "MISHAP")]	SINGLE_EVENT	What is the probability of any mishap?
MISHAP_MIN: Pmin=? [F CE_RF]	SINGLE_EVENT	What is the minimum probability of a mishap from hazard RF?
MISHAP_MAX: Pmax=? [F CE_RF]	SINGLE_EVENT	What is the maximum probability of a mishap from hazard RF?
MISHAP_UNHANDLED: S=? [?]	SINGLE_EVENT	What is the steady-state (long-run) probability of a mishap from an unhandled hazard RF?
MISHAP_HANDLED: S=? [RFp=mis]	SINGLE_EVENT	What is the steady-state (long-run) probability of a mishap from a handled hazard RF?
MISHAP_ANY: S=? ["MISHAP"]	SINGLE_EVENT	What is the steady-state (long-run) probability of any mishap?
MISHAP_COND: filter(max, P=? [F RF], {1})	MULTI_EVENT	What is the (min,avg,max) probability of mishap RF under the condition of critical event 1 occurred beforehand?
EXPECTED_SEVERITY_Q: R{"risk_sev"}=? [S]	NON_PARAMETRIC	What is the average expected severity in the long-run?
MIN_WEIGHT_ACC: R{"RF"}min=? [C<=t]	WEIGHT_PARAM	What is the minimum expected RF accumulated over the first t steps?
MAX_WEIGHT_ACC: R{"RF"}max=? [C<=t]	WEIGHT_PARAM	What is the maximum expected RF accumulated over the first t steps?
MIN_WEIGHT_TOT: R{"RF"}min=? [C]	WEIGHT_PARAM	What is the minimum expected RF in total? (NB: Not available for MDPs!)

cont'd on next page

Table A.3: Comprehensive list of properties for controller verificaton (cont'd)

PropertyId: Property	**Tags**	**Description of Property**
MAX_WEIGHT_TOT: R{"RF"}max=? [C]	WEIGHT_PARAM	What is the maximum expected RF in total? (Note: Remove any non-zero reward end components from the transition model!)
CHK_SAF_PERF: multi(R{"risk_sev"}<=s [C<=t], P<=p [F "ANY"])	QUERY	Decide whether there is a controller below an expected severity s and a hazard probability p.
MAX_PROD: multi(R{"prod"}max=? [C], R{"risk_sev"}<=s [C])	TOTAL	Select a controller that maximises productivity while staying below an expected cumulative exposure to severe injuries of p.
MAX_BOUNDED: multi(R{"REW1"}max=? [C<=t], R{"REW2"}<=p [C<=t])	CTR_SYNTHESIS	Select a controller that maximises RF while the expected cumulative 1 stays below p within t.
OPT_RISK_PERF: multi(R{"prod"}max=? [C], R{"risk_sev"}<=s [C], R{"risk_level"}<=r [C])	TOTAL	Select a controller that maximises productivity within the two bounds of risk level r and expected severity s.
MAX_POT_BND_PROD: multi(R{"potential"}max=? [C<=t], R{"prod"}>=p [C<=t])	CUMULATIVE	Select the controller with the maximum hazard mitigation potential while maintaining a minimum productivity of p in the first t steps.
DEADLOCK_IN_FINAL: A [G (!"deadlock" | "FINAL")]	MODEL_TESTING	Is every deadlock state a 'final' state? (Does our definition of 'final' include all deadlock states?)
DEADLOCK_IN_FINAL2: E [F ("deadlock" \& !"FINAL")]	MODEL_TESTING	Are all deadlocking states final? (Show a witness if violated.)
FINAL_IN_DEADLOCK: A [G ("FINAL" => ("deadlock" | (X "deadlock")))]	MODEL_TESTING	Are all 'final' states also (intermediate) deadlock states?
INFINITE_PATH: E [G !"FINAL"]	MODEL_TESTING	Is there a path not eventually leading to the final state?
INFINITE_PATH_VAR: E [F "hFINAL_CUSTOM" => (F !"hFINAL_CUSTOM" => (F "hFINAL_CUSTOM"))]	MODEL_TESTING	Can the model iterate through several cycles (indicated by `hFINAL_CUSTOM`) without reaching and stopping at `FINAL`?
INIFINITE_PATH_THROUGH_FINAL: E [F "FINAL" \& (F !"deadlock")]	MODEL_TESTING	Is there a deadlock-free (an infinite) path passing the final state?
FINALS_DO_ALWAYS_DEADLOCK: A [G ("FINAL" => (F "deadlock"))]	MODEL_TESTING	Do all reachable 'final' states (eventually) lead to deadlock states?
FINALS_CAN_ALWAYS_DEADLOCK: A [F "FINAL" => ((F "deadlock") | (G "deadlock"))]	MODEL_TESTING	Can we always reach a deadlocking 'final' state? (Is there a controller avoiding cycles through non-zero rewards?)

cont'd on next page

Table A.3: Comprehensive list of properties for controller verificaton (cont'd)

PropertyId: Property	**Tags**	**Description of Property**
NON_DEADLOCKING_FINALS: `A [ F "FINAL" => !(F "deadlock") ]`	MODEL_TESTING	Can we always reach a non-deadlocking 'final' state?
FINALS_WITH_INF_CONT: `filter(forall, A [ G "deadlock" ], "FINAL" \& !"init")`	MODEL_TESTING	Find 'final' states from which we can (erroneously) continue perpetually.
FINALS_WITH_INF_CONT2: `filter(exists, E [ !(F "deadlock") ], "FINAL")`	MODEL_TESTING	Are there reachable 'final' states with at least one deadlock-free continuation?
INITIAL_FINAL_INITIAL: `filter(exists, E [ F "init" ], "FINAL")`	MODEL_TESTING	Is there a path from an initial state through a 'final' state and returning to an 'initial' state?
FINAL_XOR_INITIAL: `filter(exists, true, "FINAL" \& "init")`	MODEL_TESTING	Does the 'final' state overlap with the 'initial' state? (Should be empty here.)
FINALS_WITHOUT_ANY_DEADLOCKS: `filter(exists, A [ G !"deadlock" ], "FINAL")`	MODEL_TESTING	Are there reachable 'final' states from which all paths are deadlock-free?

List of Tables

List of Figures

Bibliography

Hoare, Tony (1985). *Communicating Sequential Processes*. Int. Series in Comp. Sci. Prentice-Hall. URL: http://www.usingcsp.com.

Jones, Cliff B. (1986). *Systematic Program Development Using VDM*. Prentice-Hall.

Spivey, J. M. (1989). "An Introduction to Z and Formal Specification". In: *IET Software Engineering Journal* 4.1, pp. 40–50. ISSN: 0268-6961. DOI: 10.1049/sej.1989.0006.

Jackson, Michael A. (2001). *Problem Frames*. Harlow: Addison-Wesley.

Letier, Emmanuel (2001). "Reasoning about Agents in Goal-oriented Requirements Engineering". Thèse de Doctorat en Sciences Appliquées. Université Catholique de Louvain. URL: https://dial.uclouvain.be/pr/boreal/object/boreal:5139/datastream/PDF_01/view.

Leveson, Nancy G. (2004). "A new accident model for engineering safer systems". In: *Safety Science* 42.4, pp. 237–70. ISSN: 0925-7535. DOI: 10.1016/s0925-7535(03)00047-x.

Broy, Manfred (2005). "Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures – The JANUS Approach". In: *Engineering Theories of Software Intensive Systems*. Ed. by Broy M. et al. Dordrecht: Springer, pp. 47–81. DOI: 10.1007/1-4020-3532-2_2.

Kwiatkowska, Marta, Gethin Norman, and David Parker (2007). "Stochastic Model Checking". In: *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM)*. Ed. by M. Bernardo and J. Hillston. Vol. 4486. LNCS. Springer, pp. 220–70. DOI: 10.1007/978-3-540-72522-0_6.

Baier, Christel and Joost-Pieter Katoen (June 11, 2008). *Principles of Model Checking*. Cambridge, Mass, USA: MIT Press.

Lamsweerde, Axel van (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Chichester: Wiley.

Abrial, Jean-Raymond (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge: Cambridge University Press.

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

Roscoe, A. William (2010). *Understanding Concurrent Systems*. London: Springer. DOI: 10.1007/978-1-84882-258-0.

Leveson, Nancy G. (2012). *Engineering a Safer World: Systems Thinking Applied to Safety*. Engineering Systems. Cambridge, Mass.: MIT Press. DOI: 10.7551/mitpress/8179.001.0001.

Friedenthal, Sanford, Alan Moore, and Rick Steiner (2014). *A Practical Guide to SysML: The Systems Modeling Language*. 3rd ed. Burlington, Mass: Morgan Kaufmann. DOI: 10.1016/C2010-0-66331-0.

Gleirscher, Mario (2014). "Behavioral Safety of Technical Systems". Dissertation. Technische Universität München. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20141120-1221841-0-1.

Sanger, Terence D. (2014). "Risk-Aware Control". In: *Neural Computation* 26.12, pp. 2669–2691. DOI: 10.1162/neco_a_00662.

Ericson, Clifton A. (2015). *Hazard Analysis Techniques for System Safety*. 2nd ed. Hoboken, N.J.: Wiley.

ISO/TS 15066 (2016). *Robots and robotic devices – Collaborative robots*. Standard. Robotic Industries Association (RIA). URL: https://www.iso.org/standard/62996.html.

Ruijter, A. de and F. Guldenmund (2016). "The bowtie method: A review". In: *Safety Science* 88, pp. 211–218. DOI: 10.1016/j.ssci.2016.03.001.

Gleirscher, Mario (2017). "Run-Time Risk Mitigation in Automated Vehicles: A Model for Studying Preparatory Steps". In: *Formal Verification of Autonomous Vehicles (FVAV), 1st iFM Workshop on*. Ed. by L. Bulwahn, M. Kamali, and S. Linker. EPTCS, pp. 75–90. DOI: 10.4204/eptcs.257.8.

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

— (2017b). "From Hazard Analysis to Hazard Mitigation Planning: The Automated Driving Case". In: *NASA Formal Methods (NFM), 9th Int. Symp.* Ed. by C. Barrett et al. Vol. 10227. LNCS. Berlin/New York: Springer. DOI: 10.1007/978-3-319-57288-8_23.

Gleirscher, Mario (2018). "Strukturen für die Gefahrenerkennung und -behandlung in autonomen Maschinen". In: *Beiträge zu einer Systemtheorie Sicherheit*. Ed. by Jürgen Beyerer and Petra Winzer. acatech DISKUSSION. München: Herbert Utz Verlag. Chap. 8.4, pp. 154–167.

Machin, Mathilde et al. (2018). "SMOF – A Safety MOnitoring Framework for Autonomous Systems". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 48.5, pp. 702–715. DOI: 10.1109/tsmc.2016.2633291.

Gleirscher, Mario, Simon Foster, and Jim Woodcock (2019). "New Opportunities for Integrated Formal Methods". In: *ACM Computing Surveys* 52 (6)., 117:1–117:36. DOI: 10.1145/3357231. arXiv: 1812.10103 [cs.SE].

Gleirscher, Mario (2020). "YAP: Tool Support for Deriving Safety Controllers from Hazard Analysis and Risk Assessments". In: *Formal Methods for Autonomous Systems (FMAS), 2nd Workshop*. Ed. by Matt Luckuck and Marie Farrell. Vol. 329. EPTCS. Open Publishing Association, pp. 31–47. DOI: 10.4204/EPTCS.329.4. arXiv: 2012.01176 [cs.SE, cs.RO].

Gleirscher, Mario and Radu Calinescu (2020). "Safety Controller Synthesis for Collaborative Robots". In: *Engineering of Complex Computer Systems (ICECCS), 25th Int. Conf., Singapore*. Ed. by Yi Li and Alan Liew. ACM, pp. 83–92. DOI: 10.1109/ICECCS51672.2020.00017. arXiv: 2007.03340 [cs.RO cs.SE cs.SY eess.SY].

Gleirscher, Mario, Radu Calinescu, and Jim Woodcock (2021). "Risk Structures: A Design Algebra for Risk-Aware Machines". In: *Formal Aspects of Computing.*, pp. 1–40. DOI: 10.1007/s00165-021-00545-4. arXiv: 1904.10386 [cs.SE].

Index