Klever Documentation

ISP RAS

Aug 26, 2022

CONTENTS

Klever is a software verification framework that aims at automated checking of programs developed in the GNU C programming language against a variety of requirements using software model checkers. You can learn more about Klever at the project site.

CHAPTER

ONE

CONTENTS

1.1 Deployment

Klever does not support standard deployment means because it consists of several components that may require complicating setup, e.g. configuring and running a web service with a database access, running system services that perform some preliminary actions with superuser rights, etc. Also, Klever will likely always require several specific addons that can not be deployed in a normal way. Please, be ready to spend quite much time if you follow this instruction first time.

1.1.1 Hardware Requirements

We recommend following hardware to run Klever:

- x86-64 CPU with 4 cores
- 16 GB of memory
- 100 GB of free disk space

We do not guarantee that Klever will operate well if you will use less powerful machines. Increasing specified hardware characteristics in 2-4 times can reduce total verification time very considerably. To generate *Klever Build Bases* for large programs, such as the Linux kernel, you need 3-5 times more free disk space.

1.1.2 Software Requirements

Klever deployment is designed to work on:

- Debian 11.
- Ubuntu 18.04 and Ubuntu 20.04.
- openSUSE 15.3.

You can try it for other versions of these distributions, as well as for their derivatives on your own risk.

To deploy Klever you should clone its Git repository (a path to a directory where it is cloned is referred to as *$KLEVER_SRC*):

```
$ git clone --depth 1 --branch v3.7 https://forge.ispras.ru/git/klever.git
```

Note: You can try the latest version from the master branch as well as other versions at your own risk.

Note: Alternatively one can use https://github.com/ldv-klever/klever.git.

Then you need to install all required dependencies.

First of all it is necessary to install packages listed at the following files:

- Debian/Ubuntu - `klever/deploys/conf/debian-packages.txt` from *$KLEVER_SRC*.
- openSUSE - `klever/deploys/conf/opensuse-packages.txt` from *$KLEVER_SRC*.

Then you need to install Python 3.10 or higher and a corresponding development package. If your distribution does not have them you can get them from:

- Debian 11 and Ubuntu 20.04 - here.
- Ubuntu 18.04 - here.
- openSUSE 15.3 - here.

To install required Python packages we recommend to create a virtual environment using installed Python. For instance, you can run following commands within *$KLEVER_SRC*:

```
$ /usr/local/python3.10-klever/bin/python3 -m venv venv
$ source venv/bin/activate
```

To avoid some unpleasant issues during installation we recommend to upgrade PIP and associated packages:

```
$ pip install --upgrade pip wheel setuptools setuptools_scm
```

Note: Later we assume that you are using the Klever Python virtual environment created in the way described above.

Then you need to install Python packages including the Klever one:

- For production use it is necessary to run the following command within *$KLEVER_SRC*:

  ```
  $ pip install -r requirements.txt .
  ```

  Later to upgrade the Klever Python package you should run:

  ```
  $ pip install --upgrade -r requirements.txt .
  ```

- If one is going to develop Klever one should install Klever Python package in the *editable* mode (with flag *-e*). To do it, run the following command within *$KLEVER_SRC*:

  ```
  $ pip install -r requirements.txt -e .
  ```

  In this case the Klever Python package will be updated automatically, but you may still need to upgrade its dependencies by running the following command:

  ```
  $ pip install --upgrade -r requirements.txt -e .
  ```

Note: Removing *-r requirements.txt* from the command will install latest versions of required packages. However, it is not guaranteed that they will work well with Klever.

For Debian 11 you need to switch control groups from v2 to v1, since BenchExec does not support v2. Besides, this is relevant for any Linux distribution that is not mentioned above and that uses control groups v2. You can follow this instruction for Debian 11. In addition, you need to run "sudo update-grub" and reboot your system.

Then one has to get *Klever Addons* and *Klever Build Bases*. Both of them should be described appropriately within *Deployment Configuration File*.

Note: You can omit getting *Klever Addons* if you will use default *Deployment Configuration File* since it contains URLs for all required *Klever Addons*.

1.1.3 Klever Addons

You can provide *Klever Addons* in various forms:

- Local files, directories, archives or Git repositories.
- Remote files, archives or Git repositories.

Deployment scripts will take care of their appropriate extracting. If *Klever Addons* are provided locally the best place for them is directory `addons` within *$KLEVER_SRC* (see *Structure of Klever Git Repository*).

Note: Git does not track `addons` from *$KLEVER_SRC*.

Klever Addons include the following:

- *CIF*.
- *Frama-C (CIL)*.
- *Consul*.
- One or more *Verification Backends*.
- *Optional Addons*.

CIF

One can download CIF binaries from here. These binaries are compatible with various Linux distributions since CIF is based on GCC that has few dependencies. Besides, one can clone CIF Git repository and build CIF from source using corresponding instructions.

Frama-C (CIL)

You can get Frama-C (CIL) binaries from here. As well, you can build it from this source (branch `18.0`) which has several specific patches relatively to the mainline.

Consul

One can download appropriate Consul binaries from here. We are successfully using version 0.9.2 but newer versions can be fine as well. It is possible to build Consul from source.

Verification Backends

You need at least one tool that will perform actual verification of your software. These tools are referred to as *Verification Backends*. As verification backends Klever supports CPAchecker well. Some other verification backends are supported experimentally and currently we do not recommend to use them. You can download binaries of CPAchecker from here. In addition, you can clone CPAchecker Git or Subversion repository and build other versions of CPAchecker from source referring corresponding instructions.

Optional Addons

If you are going to solve verification tasks using VerifierCloud, you should get an appropriate client. Most likely one can use the client from the *CPAchecker verification backend*.

Note: For using VerifierCloud you need appropriate credentials. But anyway it is an optional addon, one is able to use Klever without it.

1.1.4 Klever Build Bases

In addition to *Klever Addons* one should provide *Klever Build Bases* obtained for software to be verified. *Klever Build Bases* should be obtained using Clade. All *Klever Build Bases* should be provided as directories, archives or links to remote archives. The best place for *Klever Build Bases* is the directory `build bases` within *$KLEVER_SRC* (see *Structure of Klever Git Repository*).

Note: Git does not track `build bases` from *$KLEVER_SRC*.

Note: Content of *Klever Build Bases* is not modified during verification.

1.1.5 Deployment Configuration File

After getting *Klever Addons* and *Klever Build Bases* one needs to describe them within *Deployment Configuration File*. By default deployment scripts use `klever/deploys/conf/klever.json` from *$KLEVER_SRC*. We recommend to copy this file somewhere and adjust it appropriately.

There are 2 pairs within *Deployment Configuration File* with names *Klever Addons* and *Klever Build Bases*. The first one is a JSON object where each pair represents a name of a particular *Klever addon* and its description as a JSON object. There is the only exception. Within *Klever Addons* there is *Verification Backends* that serves for describing *Verification Backends*.

Each JSON object that describes a *Klever addon* should always have values for *version* and *path*:

- *Version* gives a very important knowledge for deployment scripts. Depending on values of this pair they behave appropriately. When entities are represented as files, directories or archives deployment scripts remember versions of installed/updated entities. So, later they update these entities just when their versions change. For Git repositories versions can be anything suitable for a Git checkout, e.g. appropriate Git branches, tags or commits. In this case deployment scripts checkout specified versions first. Also, they clone or clean up Git repositories before checkout, so, all uncommitted changes will be ignored. To bypass Git checkout and clean up you can specify version *CURRENT*. In this case Git repositories are treated like directories.

- *Path* sets either a path relative to *$KLEVER_SRC* or an absolute path to entity (binaries, source files, configurations, etc.) or an entity URL.

For some *Klever Addons* it could be necessary to additionally specify *executable path* or/and *python path* within *path* if binaries or Python packages are not available directly from *path*. For *Verification Backends* there is also *name* with value *CPAchecker*. Keep this pair for all specified *Verification Backends*.

Besides, you can set *copy .git directory* and *allow use local Git repository* to *True*. In the former case deployment scripts will copy directory `.git` if one provides *Klever Addons* as Git repositories. In the latter case deployment scripts will use specified Git repositories for cleaning up and checkout required versions straightforwardly without cloning them to temporary directories.

Warning: Setting *allow use local Git repository* to *True* will result in removing all your uncommitted changes! Besides, ignore rules from, say, `.gitignore` will be ignored and corresponding files and directories will be removed!

Klever Build Bases is a JSON object where each pair represents a name of a particular *Build Base* and its description as a JSON object. Each such JSON object should always have some value for *path*: it should be either an absolute path to the directory that directly contains *Build Base*, or an absolute path to the archive with a *Build Base*, or a link to the remote archive with a *Build Base*. Particular structure of directories inside such archive doesn't matter: it is only required that there should be a single valid *Build Base* somewhere inside. In `job.json` you should specify the name of the *Build Base*.

Note: You can prepare multiple *deployment configuration files*, but be careful when using them to avoid unexpected results due to tricky intermixes.

Note: Actually there may be more *Klever Addons* or *Klever Build Bases* within corresponding locations. Deployment scripts will consider just described ones.

1.1.6 Structure of Klever Git Repository

After getting *Klever Addons* and *Klever Build Bases* the Klever Git repository can look as follows:

```
$KLEVER_SRC
├── addons
│   ├── cif-1517e57.tar.xz
│   ├── consul
│   ├── CPAchecker-1.6.1-svn ea117e2ecf-unix.tar.gz
│   ├── CPAchecker-35003.tar.xz
│   ├── toplevel.opt.tar.xz
│   └── ...
├── build bases
│   ├── linux-3.14.79.tar.xz
│   └── linux-4.2.6
│       ├── allmodconfig
│       └── defconfig
│
```

```
└── ...
```

1.1.7 Deployment Variants

There are several variants for deploying Klever:

Local Deployment

Warning: Do not deploy Klever at your workstation or valuable servers unless you are ready to lose some sensitive data or to have misbehaved software.

Warning: Currently deployment on Fedora makes the *httpd_t* SELinux domain permissive, which may negatively impact the security of your system.

To accomplish local deployment of Klever you need to choose an appropriate mode (one should select *development* only for development purposes, otherwise, please, choose *production*) and to run the following command within *$KLEVER_SRC*:

```
$ sudo venv/bin/klever-deploy-local --deployment-directory $KLEVER_DEPLOY_DIR install production
```

Note: Absolute path to `klever-deploy-local` is necessary due to environment variables required for the Klever Python virtual environment are not passed to sudo commands most likely.

Note: You should install Klever Python package in the editable mode in case of the development mode (*Software Requirements*). Otherwise, some functionality may not work as intended.

After successful installation one is able to *update* Klever multiple times to install new or to update already installed *Klever Addons* and *Klever Build Bases*:

```
$ sudo venv/bin/klever-deploy-local --deployment-directory $KLEVER_DEPLOY_DIR update production
```

If you need to update Klever Python package itself (e.g. this may be necessary after update of *$KLEVER_SRC*), then you should execute one additional command prior to the above one:

```
$ pip install --upgrade .
```

This additional command, however, should be skipped if Klever Python package was installed in the *editable* mode (with flag -e) unless you need to to upgrade Klever dependencies. In the latter case you should execute the following command prior updating Klever:

```
$ pip install --upgrade -e .
```

To *uninstall* Klever you need to run:

```
$ sudo venv/bin/klever-deploy-local --deployment-directory $KLEVER_DEPLOY_DIR uninstall␣
↪production
```

A normal sequence of actions for *Local Deployment* is the following: *install* → *update* → *update* → … → *update* → *uninstall*. In addition, there are several optional command-line arguments which you can find out by running:

```
$ klever-deploy-local --help
```

We strongly recommend to configure your file indexing service if you have it enabled so that it will ignore content of *$KLEVER_DEPLOY_DIR*. Otherwise, it can consume too much computational resources since Klever manipulates files very extensively during its operation. To do this, please, refer to an appropriate user documentation.

Troubleshooting

If something went wrong during installation, you need to uninstall Klever completely prior to following attempts to install it. In case of ambiguous issues in the development mode you should try to remove the virtual environment and to create it from scratch.

OpenStack Deployment

Note: Although we would like to support different OpenStack environments, we tested *OpenStack Deployment* just for the ISP RAS one.

Additional Software Requirements

To install additional packages required only by OpenStack deployment scripts you need to execute the following command:

```
$ pip install -r requirements-openstack.txt ".[openstack]"
```

Note: If in the previous step you installed Klever package with the *-e* argument, then you should use it here as well (i.e. execute *pip install -r requirements-openstack.txt -e ".[openstack]"*).

Supported Options

OpenStack Deployment supports 2 kinds of entities:

- *Klever Base Image* - with default settings this is a Debian 11 OpenStack image with installed Klever dependencies. Using *Klever Base Image* allows to substantially reduce a time for deploying *Klever Instances*.
- *Klever Instance* - an OpenStack instance, either for development or production purposes. For development mode many debug options are activated by default.

Almost all deployment commands require you to specify a path to a private SSH key and your OpenStack username:

```
$ klever-deploy-openstack --os-username $OS_USERNAME --ssh-rsa-private-key-file $SSH_RSA_
↪PRIVATE_KEY_FILE create instance
```

For brevity they are omitted from the following examples.

In addition to command-line arguments mentioned above and below, there are several optional command-line arguments which you can find out by running:

```
$ klever-deploy-openstack --help
```

Klever Base Image

For *Klever Base Image* you can execute actions *show*, *create* and *remove*. The common workflow for *Klever Base Image* is *create* → *remove*, e.g.:

```
$ klever-deploy-openstack create image
```

Unless specified, name *Klever Base vN* (where *N* is 1 plus a maximum of 0 and *vi*) is used for new *Klever Base Image*. Besides, deployment scripts overwrites file `klever/deploys/conf/openstack-base-image.txt` with this name so that new instances will be based on the new *Klever Base Image*. To force other users to switch to the new *Klever Base Image* you need to commit changes of this file to the repository.

Klever Instance

For *Klever Instance* you can execute actions *show*, *create*, *update*, *resize*, *share*, *hide*, *ssh* and *remove*. Basically you should perform actions with *Klever Instance* in the following order: *create* → *update/resize* → *update/resize* → … → *update/resize* → *remove* like for *Local Deployment*, e.g.:

```
$ klever-deploy-openstack create instance
```

By default Klever is deployed in the production mode, but you can change this with the *--mode* command-line argument:

```
$ klever-deploy-openstack --mode development create instance
```

By using *resize* you can increase/decrease occupied computational resources and, thus, either speed up the verification process considerably or do not waste always necessary CPU cores and memory. You should take into account that it is enough to specify only the number of required CPU cores, e.g. 1, 2, 4, etc., since there is a rule of thumb to compute the appropriate size of memory (4x of the number of CPU cores) while disk space can not be modified during resizing. In addition, between creation and removal of *Klever Instance* you can also *share/hide* it for/from the outside world and open an SSH connection to it.

Note: You should be especially careful with user credentials when sharing *Klever Instance*.

By default name for created *Klever Instance* is a concatenation of *$OS_USERNAME*, "klever", and the mode used (development or production), e.g. *petrov-klever-development*.

Multiple Klever Instances

You can also create a specified number of OpenStack instances for performing various experiments by using the *--instances* command-line argument. In this mode you can only execute actions *show*, *create*, *update* and *remove*. The normal workflow for *Multiple Klever Instances* is the same as for *Klever Instance*, e.g.:

```
$ klever-deploy-openstack --instances $INSTANCES create instance
```

1.2 Tutorial

This tutorial describes a basic workflow of using Klever. We assume that you deployed Klever *locally* on Debian 11 in the production mode with default settings. In addition, we assume that your username is **debian** and your home directory is **/home/debian**[1].

1.2.1 Preparing Build Bases

After a successful deployment of Klever you need to get a *build base*[2]. This tutorial treats just build bases for Linux kernel loadable modules since the publicly available version of Klever has only experimental support for verification of other software. You should not expect that Klever supports all versions and configurations of the Linux kernel well. There is a big list of things to do in this direction.

Below we consider as an example a build base for verification of kernel loadable modules of Linux 5.5.19 (architecture *x86_64*, configuration *allmodconfig*). You can download the archive of the target build base prepared in advance from here. Let's assume that you unpack this archive into directory **/home/debian/build-base-linux-5.5.19-x86_64-allmodconfig** so that there should be file *meta.json* directly at the top level in that directory. Besides, you can use in a similar way build bases prepared for:

- Linux 5.10.120
- Linux 5.17.13

If you want to prepare the build base yourself, we recommend to do this on the same machine where you deployed Klever since it already contains everything necessary. You can try to execute similar steps for other versions and configurations of the Linux kernel at your own risks. To build other versions of the Linux kernel you may need appropriate versions of GCC as well as other build time prerequisites.

To prepare the target build base from scratch you can follow the next steps:

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.5.19.tar.xz
$ tar -xJf linux-5.5.19.tar.xz
$ cd linux-5.5.19
$ make allmodconfig
$ source $KLEVER_SRC/venv/bin/activate
$ clade -w ~/build-base-linux-5.5.19-x86_64-allmodconfig -p klever_linux_kernel --cif
↪$KLEVER_DEPLOY_DIR/klever-addons/CIF/bin/cif make -j8 modules
```

Then you will need to wait for quite a long period of time depending on the performance of your machine (typically several hours). If everything will go well, you will have the target build base in directory **/home/debian/build-base-linux-5.5.19-x86_64-allmodconfig**.

[1] If this is not the case, you should adjust paths to build bases below respectively.

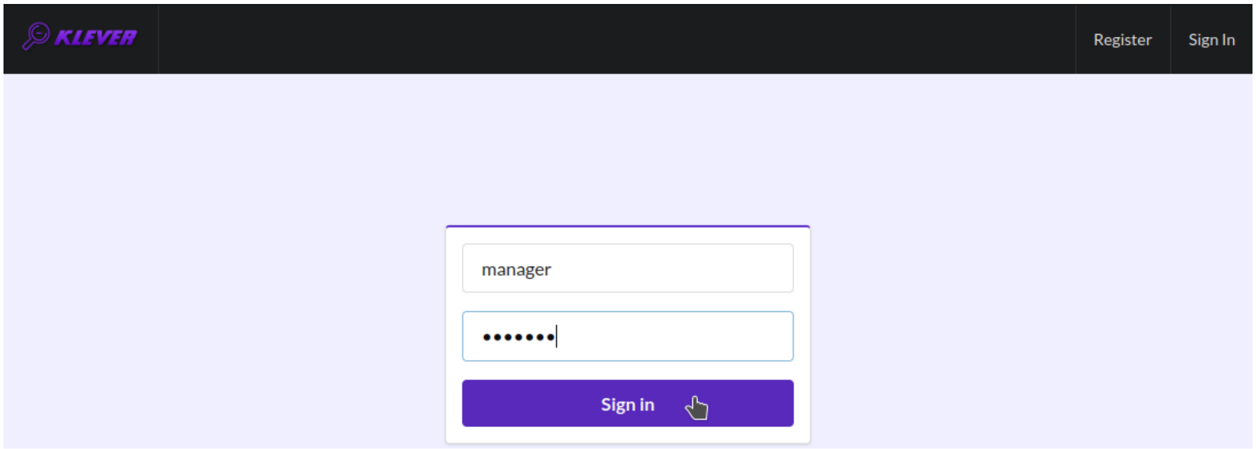
[2] Several build bases are deployed together with Klever by default, but they contain just a small subset of Linux kernel loadable modules. The corresponding Linux kernel version is 3.14.79, target architectures are x86-64, ARM and ARM64.

1.2.2 Signing in

Before performing all other actions described further in this tutorial you need to sign in using a Klever web interface:

1. Open page http://localhost:8998 in your web-browser[3].
2. Input **manager** as a username and a password and sign in (Fig. 1.1).

Then you will be automatically redirected to a *job tree* page presented in the following sections.



manager

•••••••

Fig. 1.1: Signing in

1.2.3 Starting Verification

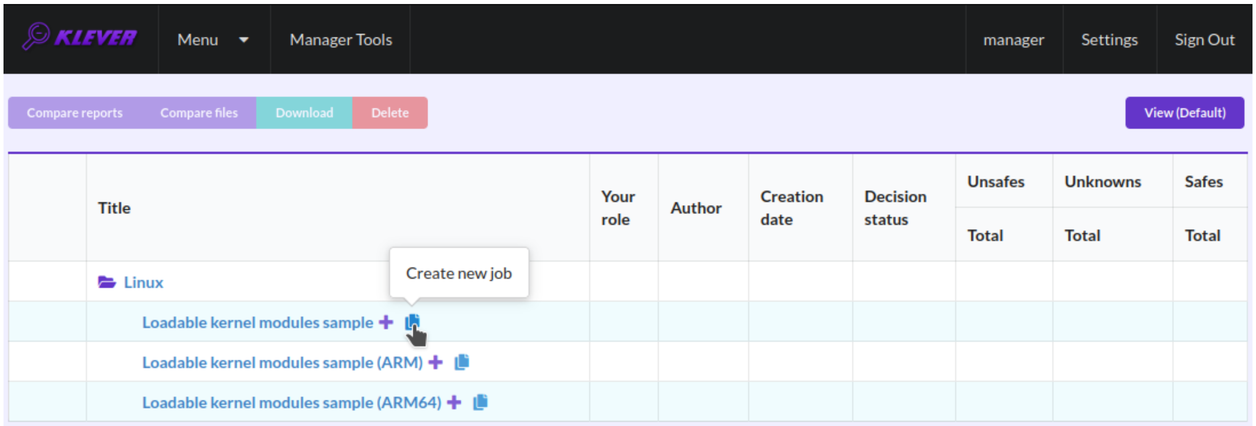
As an example we consider checking usage of clocks and memory safety in USB drivers. To start up verification you need to do as follows:

1. Start the creation of a new *job* (Fig. 1.2).
2. Specify an appropriate title and create the new job (Fig. 1.3).
3. To configure a first *job version* you need to specify (Fig. 1.4):
   - The path to the prepared build base that is **/home/debian/build-base-linux-5.5.19-x86_64-allmodconfig**.
   - Targets, e.g. USB drivers, i.e. all modules from directory **drivers/usb** in our example.
   - Specifications set *5.5* (you can see a list of all supported specification sets at the end of this section).
   - Requirement specifications to be checked, e.g. **drivers:clk.*** and **memory safety** in our example (you can see a list of all supported requirement specifications at the end of this section).
4. Press *Ctrl-S* with your mouse pointer being at the editor window to save changes.
5. Start a *decision of the job version* (Fig. 1.4).

After that Klever automatically redirects you to a *job version/decision page* that is described in detail in the following sections.

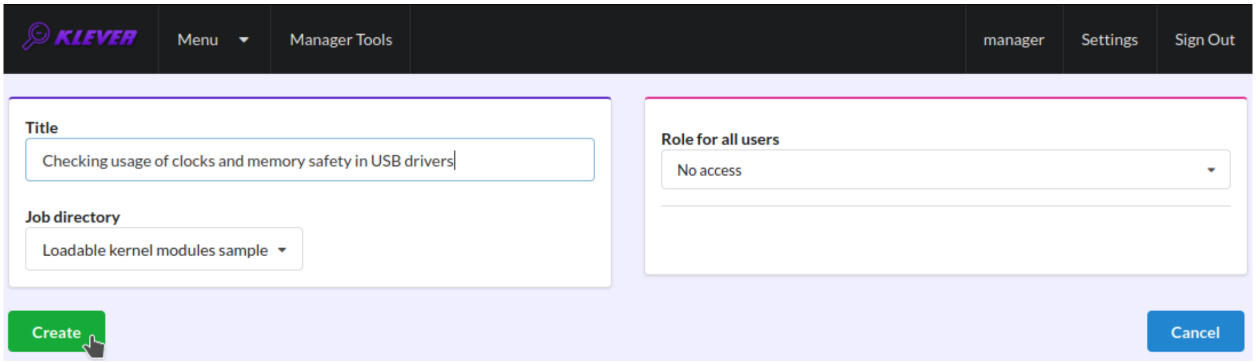
Later you can create new jobs by opening the job tree page, e.g. through clicking on the Klever logo (Fig. 1.5), and by executing steps above. You can create new jobs even when some job version is being decided, but various job versions are decided one by one by default.

[3] You can open the Klever web interface from other machines as well, but you need to set up appropriate access for that.



Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
					Total	Total	Total
Linux							
Loadable kernel modules sample							
Loadable kernel modules sample (ARM)							
Loadable kernel modules sample (ARM64)							

Fig. 1.2: Starting the creation of a new job



Title

Checking usage of clocks and memory safety in USB drivers

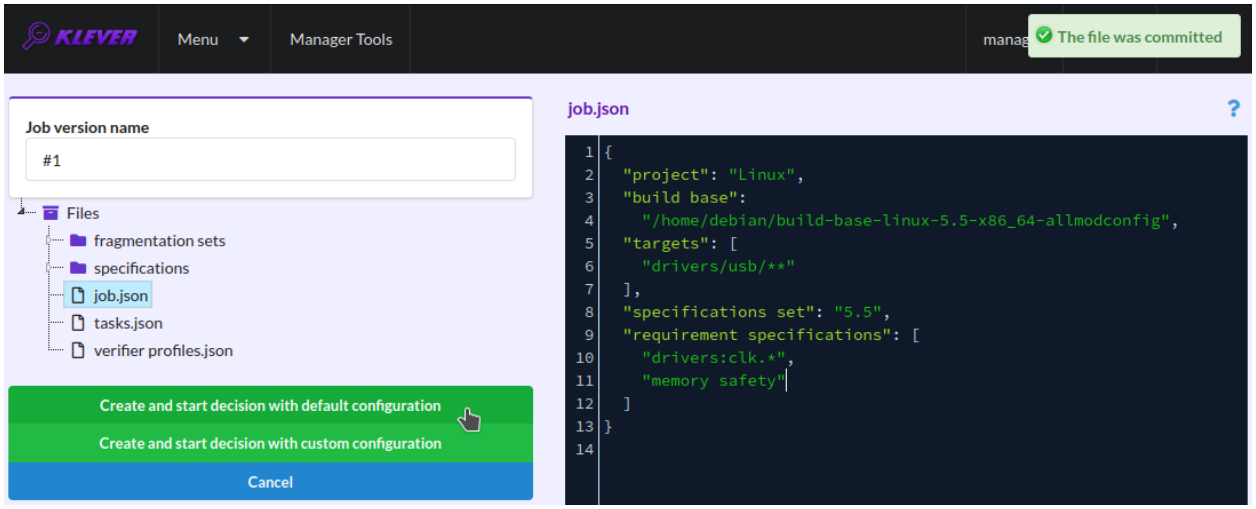
Job directory

Loadable kernel modules sample

Role for all users

No access

Fig. 1.3: The creation of the new job



Job version name

#1

job.json

```
{
  "project": "Linux",
  "build base":
    "/home/debian/build-base-linux-5.5-x86_64-allmodconfig",
  "targets": [
    "drivers/usb/**"
  ],
  "specifications set": "5.5",
  "requirement specifications": [
    "drivers:clk.*",
    "memory safety"
  ]
}
```

Fig. 1.4: Configuring the first job version and starting its decision

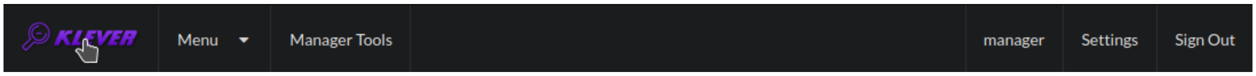


Fig. 1.5: Opening the job tree page

Below there are requirement specifications that you can choose for verification of Linux loadable kernel modules (we do not recommend to check requirement specifications which identifiers are italicised since they produce either many false alarms or there are just a few violations of these requirements at all):

1. alloc:irq
2. alloc:spinlock
3. alloc:usb lock
4. arch:asm:dma-mappingfile:///home/novikov/work/klever/docs/_build/html/tutorial.html#preparing-build-bases
5. arch:mm:ioremap
6. *block:blk-core:queue*
7. *block:blk-core:request*
8. *block:genhd*
9. *concurrency safety*
10. drivers:base:class
11. drivers:usb:core:usb:coherent
12. drivers:usb:core:usb:dev
13. drivers:usb:core:driver
14. drivers:usb:core:urb
15. drivers:usb:gadget:udc-core
16. drivers:clk1
17. drivers:clk2
18. fs:sysfs:group
19. kernel:locking:mutex
20. kernel:locking:rwlock
21. kernel:locking:spinlock
22. kernel:module
23. *kernel:rcu:update:lock bh*
24. *kernel:rcu:update:lock shed*
25. kernel:rcu:update:lock
26. *kernel:rcu:srcu*
27. *kernel:sched:completion*
28. *lib:find_next_bit*
29. *lib:idr*
30. memory safety
31. net:core:dev
32. *net:core:rtnetlink*
33. *net:core:sock*

In case of verification of the Linux kernel rather than vanilla 5.5, you may need to change a value of option **specifications set** when configuring the job version (Fig. 1.4). Klever supports following specification sets:

1. 2.6.33
2. 3.2
3. 3.14
4. 3.14-dentry-v2
5. 4.6.7
6. 4.15
7. 4.17
8. 5.5
9. 5.17

These specification sets correspond to vanilla versions of the Linux kernel. You should select such a specifications set that matches your custom version of the Linux kernel better through the trial and error method.

1.2.4 Decision Progress

At the beginning of the decision of the job version Klever indexes each new build base. This can take rather much time before it starts to generate and to decide first *tasks*[4] for large build bases. In about 15 minutes you can refresh the page and see results of decision for some tasks there. Please, note that the automatic refresh of the job version/decision page stops after 5 minutes, so you either need to refresh it through conventional means of your web browser or request Klever to switch it on back (Fig. 1.6).

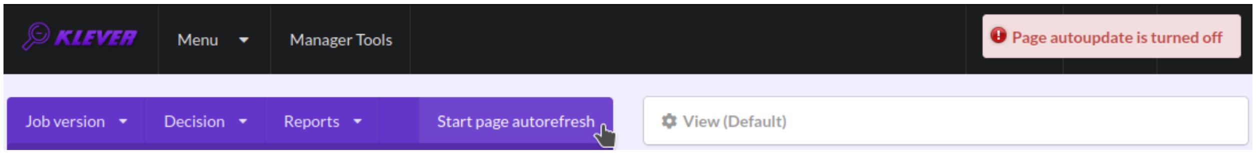
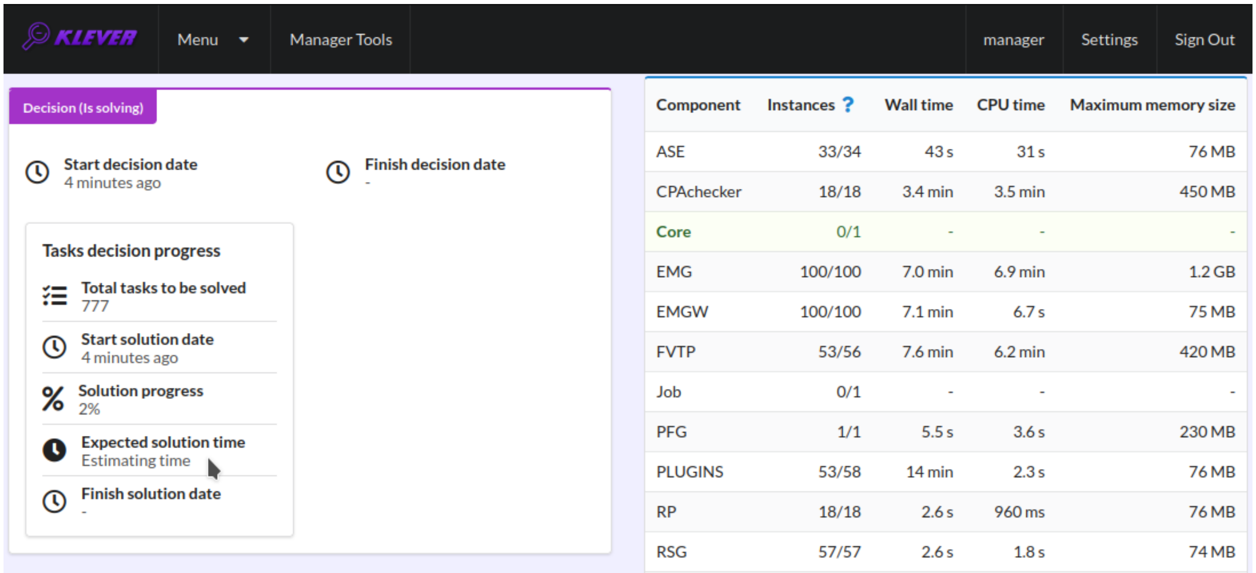


Fig. 1.6: Switching on the automatic refresh of the job version/decision page

Before the job version is eventually decided Klever estimates and provides a *decision progress* (Fig. 1.7 and Fig. 1.8). You should keep in mind that Klever collects statistics for 10% of tasks before it starts predicting an approximate remaining time for their decision. After that, it recalculates it on the base of accumulated statistics. In our example it takes about 3 hours to decide the job version completely (Fig. 1.9).

At the job tree page you can see all versions of particular jobs (Fig. 1.10) and their *decision statutes* (Fig. 1.11). Besides, you can open the page with details of the decision of the latest job version (Fig. 1.12) or the page describing the decision of the particular job version (Fig. 1.13).

[4] For the considered example each task is a pair of a Linux loadable kernel module and a requirements specification. There are 259 modules under verification and 3 requirement specifications to be checked, so there are 777 tasks in total.



Start decision date
4 minutes ago

Finish decision date
-

Tasks decision progress

Total tasks to be solved
777

Start solution date
4 minutes ago

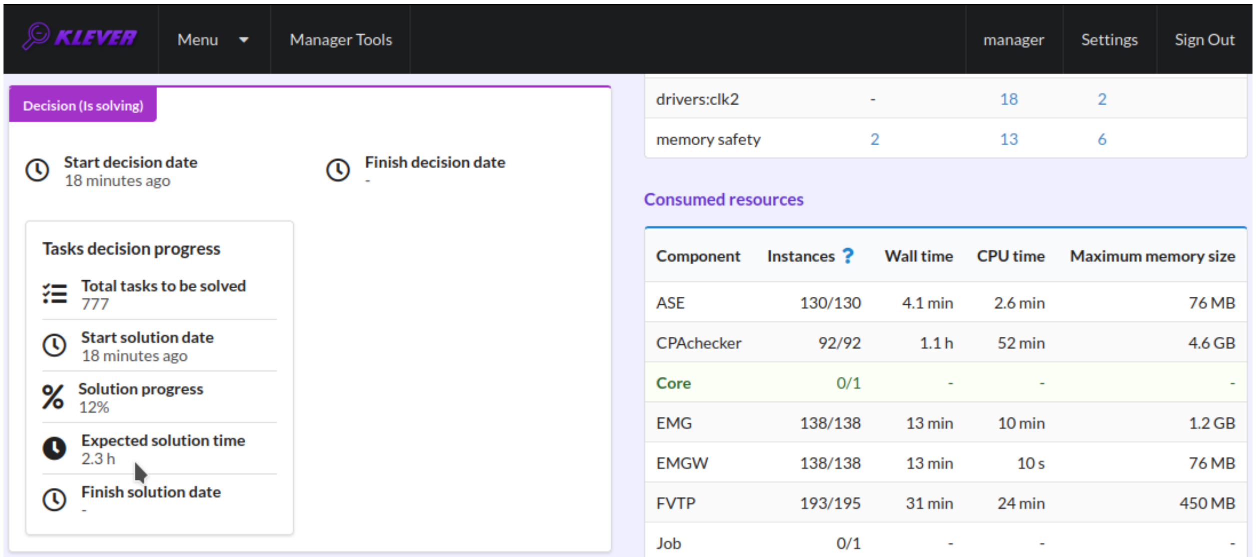
Solution progress
2%

Expected solution time
Estimating time

Finish solution date
-

Component	Instances ?	Wall time	CPU time	Maximum memory size
ASE	33/34	43 s	31 s	76 MB
CPAchecker	18/18	3.4 min	3.5 min	450 MB
Core	0/1	-	-	-
EMG	100/100	7.0 min	6.9 min	1.2 GB
EMGW	100/100	7.1 min	6.7 s	75 MB
FVTP	53/56	7.6 min	6.2 min	420 MB
Job	0/1	-	-	-
PFG	1/1	5.5 s	3.6 s	230 MB
PLUGINS	53/58	14 min	2.3 s	76 MB
RP	18/18	2.6 s	960 ms	76 MB
RSG	57/57	2.6 s	1.8 s	74 MB

Fig. 1.7: The progress of the decision of the job version (estimating a remaining time)



Start decision date
18 minutes ago

Finish decision date
-

Tasks decision progress

Total tasks to be solved
777

Start solution date
18 minutes ago

Solution progress
12%

Expected solution time
2.3 h

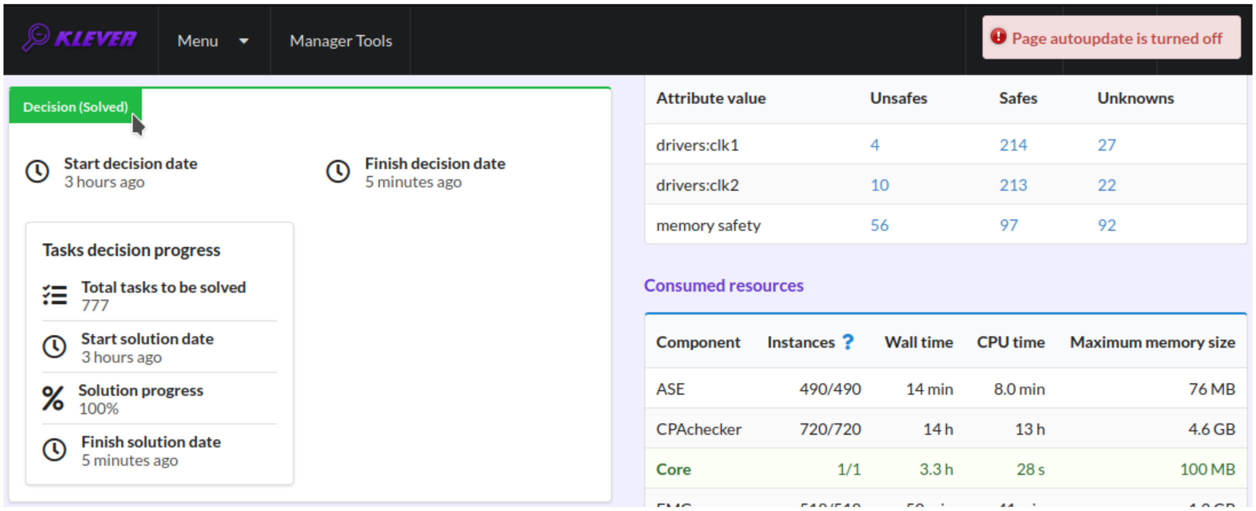
Finish solution date
-

drivers:clk2	-	18	2
memory safety	2	13	6

Consumed resources

Component	Instances ?	Wall time	CPU time	Maximum memory size
ASE	130/130	4.1 min	2.6 min	76 MB
CPAchecker	92/92	1.1 h	52 min	4.6 GB
Core	0/1	-	-	-
EMG	138/138	13 min	10 min	1.2 GB
EMGW	138/138	13 min	10 s	76 MB
FVTP	193/195	31 min	24 min	450 MB
Job	0/1	-	-	-

Fig. 1.8: The progress of the decision of the job version (the remaining time is estimated)



Start decision date
3 hours ago

Finish decision date
5 minutes ago

Tasks decision progress

Total tasks to be solved
777

Start solution date
3 hours ago

Solution progress
100%

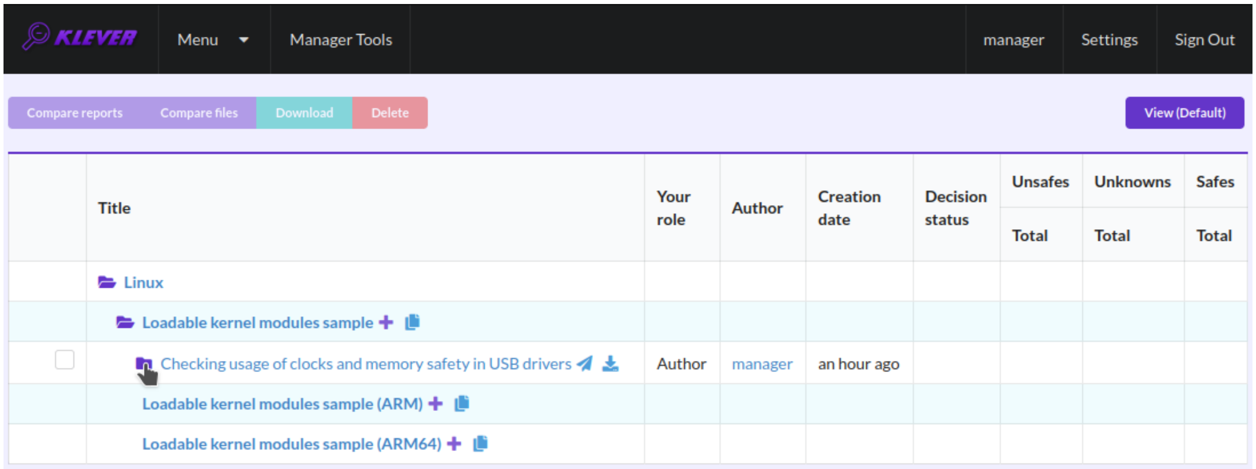
Finish solution date
5 minutes ago

Attribute value	Unsafes	Safes	Unknowns
drivers:clk1	4	214	27
drivers:clk2	10	213	22
memory safety	56	97	92

Consumed resources

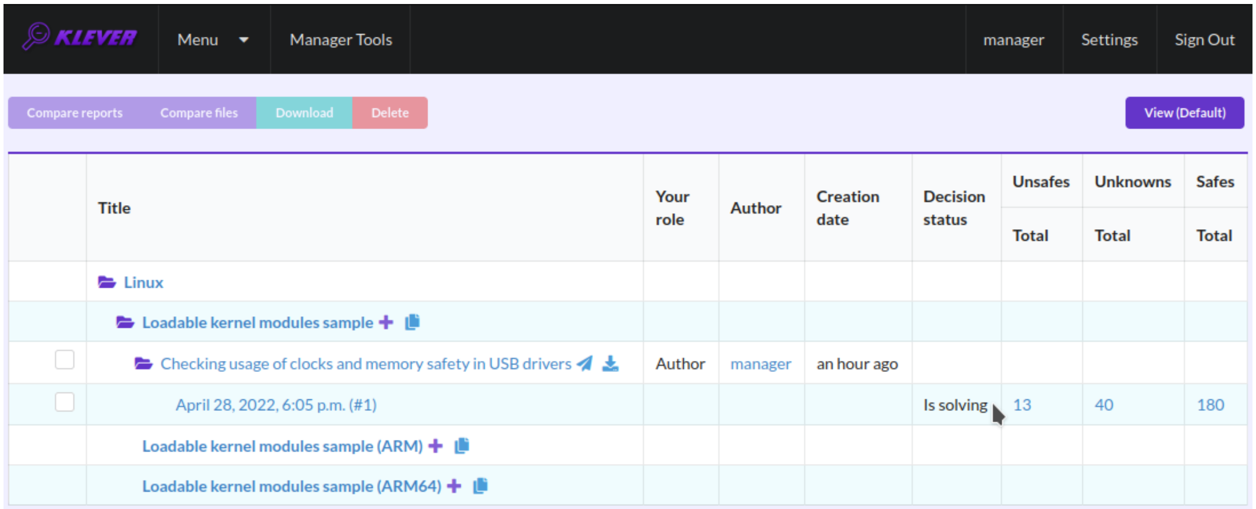
Component	Instances ?	Wall time	CPU time	Maximum memory size
ASE	490/490	14 min	8.0 min	76 MB
CPAchecker	720/720	14 h	13 h	4.6 GB
Core	1/1	3.3 h	28 s	100 MB

Fig. 1.9: The completed decision of the job version



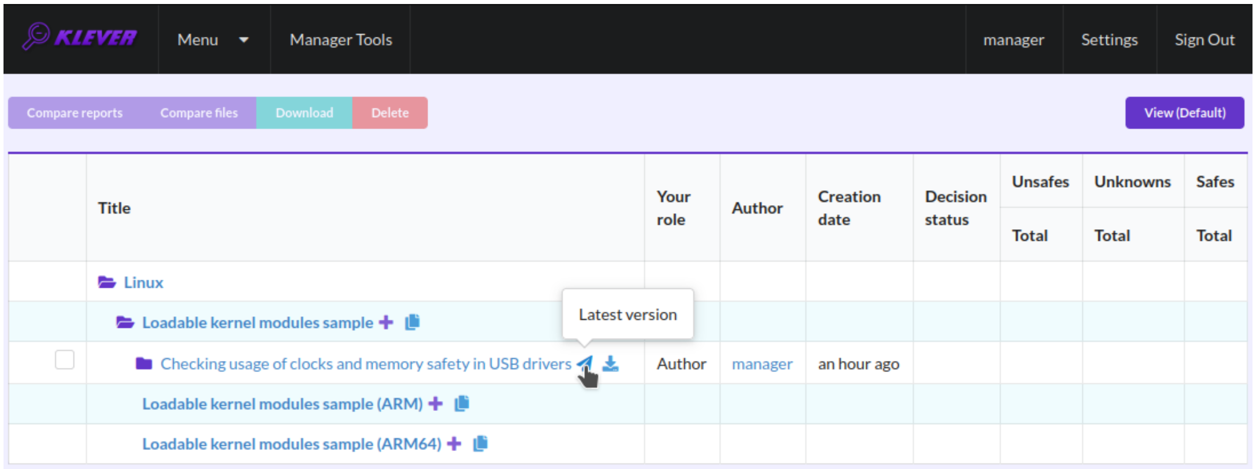
	Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
						Total	Total	Total
	Linux							
	Loadable kernel modules sample							
□	Checking usage of clocks and memory safety in USB drivers	Author	manager	an hour ago				
	Loadable kernel modules sample (ARM)							
	Loadable kernel modules sample (ARM64)							

Fig. 1.10: Showing job versions



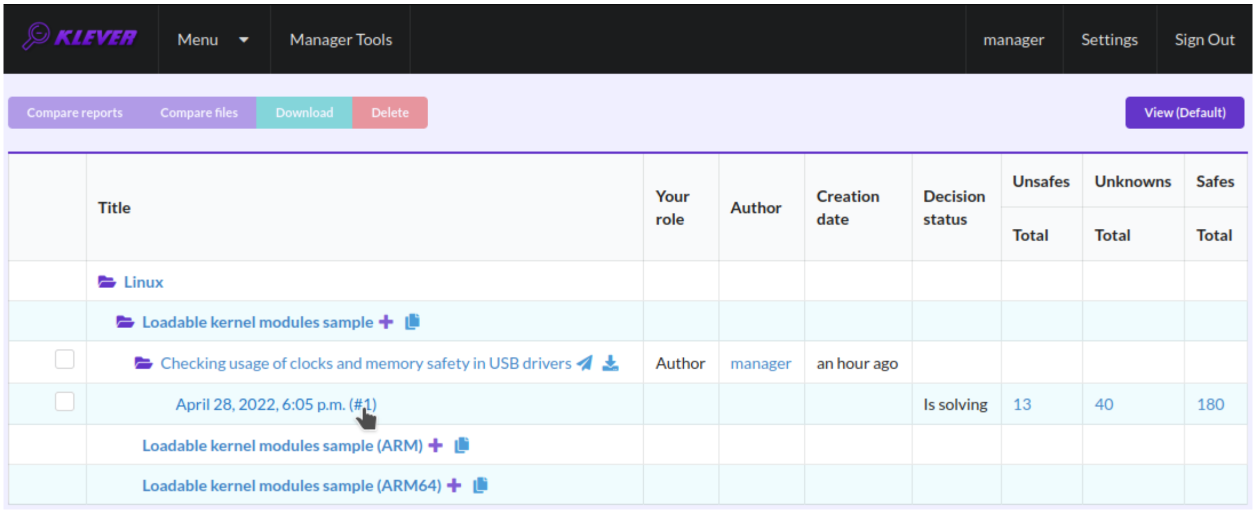
	Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
						Total	Total	Total
	Linux							
	Loadable kernel modules sample							
□	Checking usage of clocks and memory safety in USB drivers	Author	manager	an hour ago				
□	April 28, 2022, 6:05 p.m. (#1)				Is solving	13	40	180
	Loadable kernel modules sample (ARM)							
	Loadable kernel modules sample (ARM64)							

Fig. 1.11: The status of the decision of the job version



	Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
						Total	Total	Total
	Linux							
	Loadable kernel modules sample							
□	Checking usage of clocks and memory safety in USB drivers	Author	manager	an hour ago				
	Loadable kernel modules sample (ARM)							
	Loadable kernel modules sample (ARM64)							

Fig. 1.12: Opening the page with the decision of the latest job version

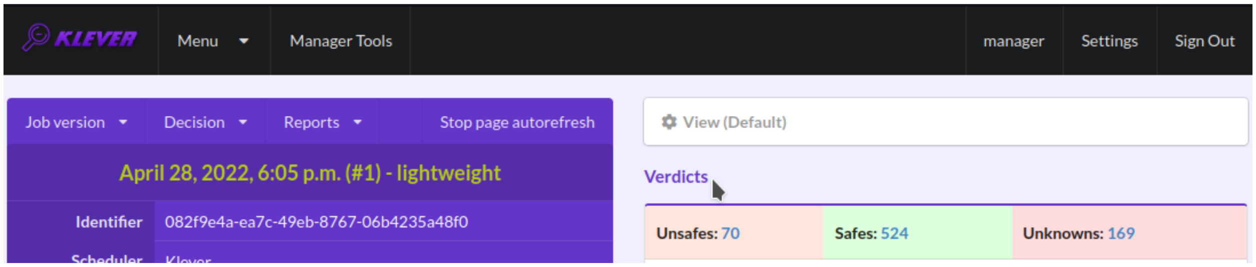


	Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
						Total	Total	Total
	Linux							
	Loadable kernel modules sample							
□	Checking usage of clocks and memory safety in USB drivers	Author	manager	an hour ago				
□	April 28, 2022, 6:05 p.m. (#1)				Is solving	13	40	180
	Loadable kernel modules sample (ARM)							
	Loadable kernel modules sample (ARM64)							

Fig. 1.13: Opening the page with the decision of the particular job version

1.2.5 Analyzing Verification Results

Klever can fail to generate and decide tasks. In this case it provides users with *unknown* verdicts, otherwise there are *safe* or *unsafe* verdicts (Fig. 1.14). You already saw the example with summaries of these verdicts at the job tree page (Fig. 1.11). In this tutorial we do not consider in detail other verdicts rather than unsafes that are either violations of checked requirements or false alarms (Fig. 1.15). Klever reports unsafes if so during the decision of the job version and you can assess them both during the decision and after its completion.



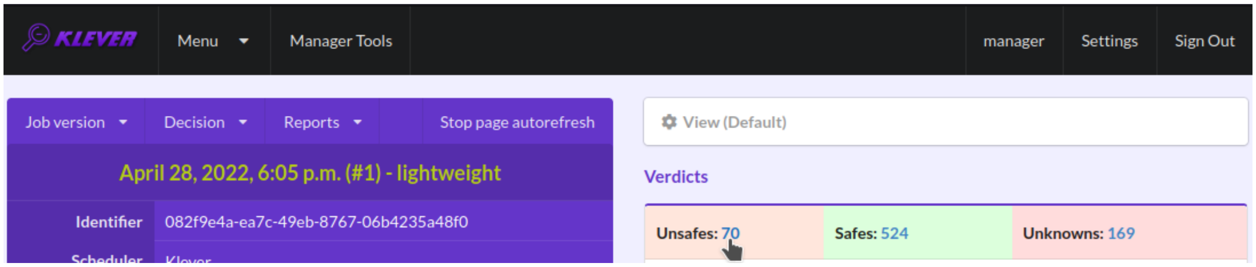
April 28, 2022, 6:05 p.m. (#1) - lightweight

Identifier 082f9e4a-ea7c-49eb-8767-06b4235a48f0

Scheduler Klever

Unsafes: 70 Safes: 524 Unknowns: 169

Fig. 1.14: Verdicts



April 28, 2022, 6:05 p.m. (#1) - lightweight

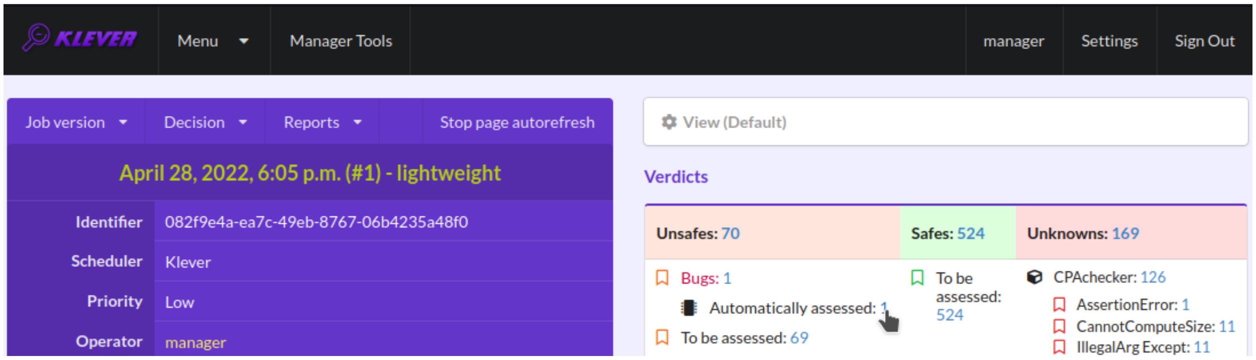
Identifier 082f9e4a-ea7c-49eb-8767-06b4235a48f0

Scheduler Klever

Unsafes: 70 Safes: 524 Unknowns: 169

Fig. 1.15: The total number of unsafes reported thus far

During assessment of unsafes experts can create marks that can match other unsafes with similar error traces (we consider marks and error traces in detail within next sections). For instance, there is a mark that matches one of the reported unsafes (Fig. 1.16). Automatic assessment can reduce efforts for analysis of verification results considerably, e.g. when verifying several versions or configurations of the same software. But experts should analyze such automatically assessed unsafes since the same mark can match unsafes with error traces that look very similar but correspond to different faults. Unsafes without marks need assessment as well (Fig. 1.17). When checking several requirement specifications in the same job, one is able to analyze unsafes just for a particular requirements specification (Fig. 1.18).



April 28, 2022, 6:05 p.m. (#1) - lightweight

Identifier 082f9e4a-ea7c-49eb-8767-06b4235a48f0

Scheduler Klever

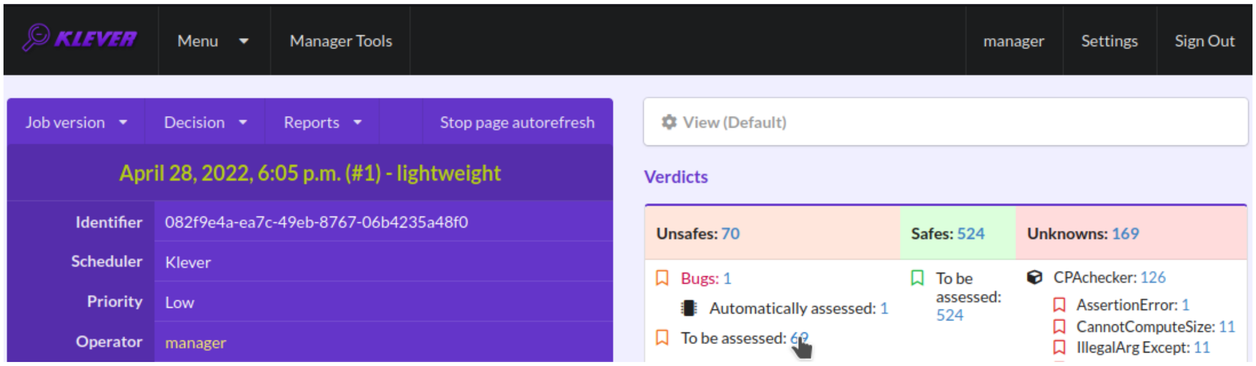
Priority Low

Operator manager

Unsafes: 70	Safes: 524	Unknowns: 169
Bugs: 1 Automatically assessed: 1 To be assessed: 69	To be assessed: 524	CPAchecker: 126 AssertionError: 1 CannotComputeSize: 11 IllegalArg Except: 11

Fig. 1.16: The total number of automatically assessed unsafes

After clicking on the links in Fig. 1.15- Fig. 1.18 you will be redirected to pages with lists of corresponding unsafes (e.g. Fig. 1.19). If there is the only element in such a list you will see an appropriate error trace immediately. For



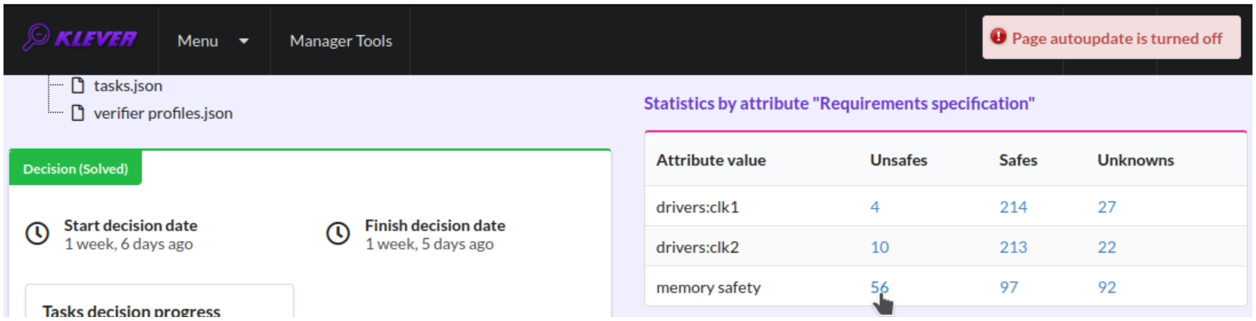
April 28, 2022, 6:05 p.m. (#1) - lightweight

Identifier	082f9e4a-ea7c-49eb-8767-06b4235a48f0
Scheduler	Klever
Priority	Low
Operator	manager

Verdicts

Unsafes: 70	Safes: 524	Unknowns: 169
Bugs: 1 Automatically assessed: 1 To be assessed: 69	To be assessed: 524	CPAchecker: 126 AssertionError: 1 CannotComputeSize: 11 IllegalArg Except: 11

Fig. 1.17: The total number of unsafes without any assessment



Start decision date
1 week, 6 days ago

Finish decision date
1 week, 5 days ago

Tasks decision progress

Statistics by attribute "Requirements specification"

Attribute value	Unsafes	Safes	Unknowns
drivers:clk1	4	214	27
drivers:clk2	10	213	22
memory safety	56	97	92

Fig. 1.18: The total number of unsafes corresponding to the particular requirements specification

further analysis we recommend clicking on an unsafe index on the left to open a new page in a separate tab (Fig. 1.20). To return back to the job version/decision page you can click on the title of the job decision on the top left (Fig. 1.21). This can be done at any page with such the link.

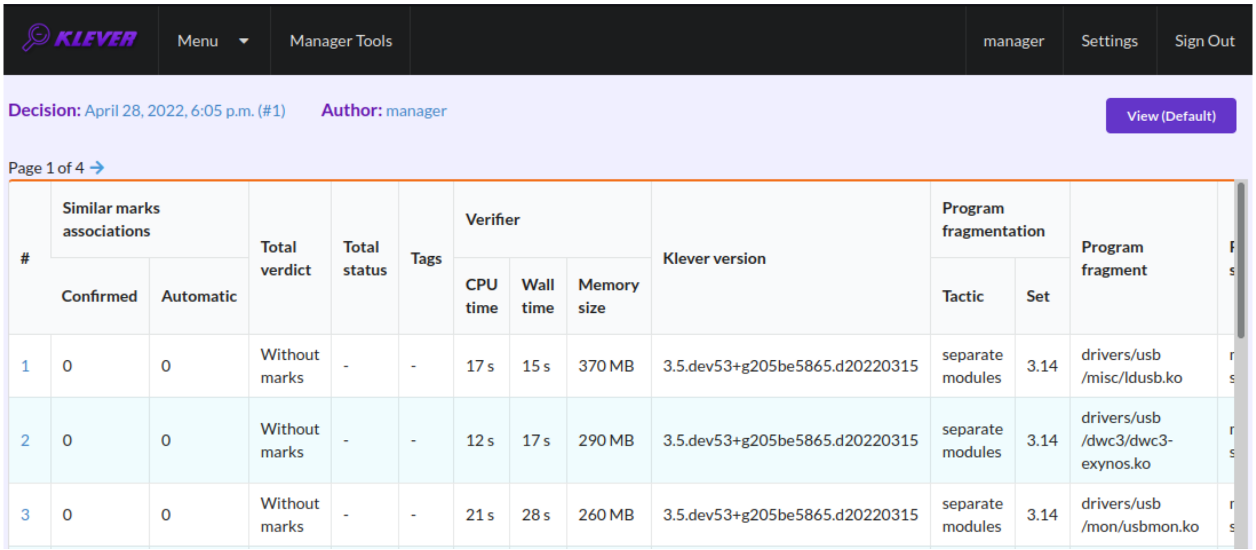
1.2.6 Analyzing Error Traces

After clicking on links within the list of unsafes like in Fig. 1.20, you will see corresponding error traces. For instance, Fig. 1.22 demonstrates an error trace example for module *drivers/usb/gadget/udc_bdc_bdc.ko* and requirements specification *drivers:clk1*.

An *error trace* is a sequence of declarations and statements in a source code of a module under verification and an *environment model* generated by Klever. Besides, within that sequence there are *assumptions* specifying conditions that a verification tool considers to be true. Declarations, statements and assumptions represent a path starting from an entry point and ending at a violation of one of checked requirements. The entry point analogue for userspace programs is function *main* while for Linux loadable kernel modules entry points are generated by Klever as a part of environment models. Requirement violations do not always correspond to places where detected faults should be fixed. For instance, the developer can omit a check for a return value of a function that can fail. As a result various issues, such as leaks or null pointer dereferences, can be revealed somewhere later.

Numbers in the left column correspond to line numbers in source files and models. Source files and models are displayed to the right of error traces. Fig. 1.22 does not contain anything at the right part of the window since there should be the environment model containing the generated *main* function but by default models are not demonstrated for users in the web interface[5]. If you click on a line number corresponding to an original source file, you will see this source file

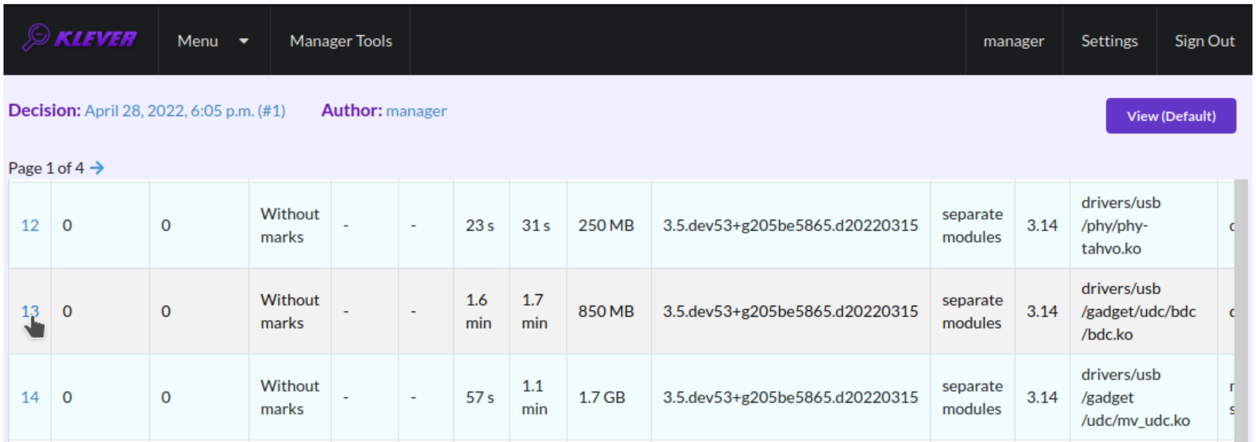
[5] If you want to see these models, you have to start the decision of the job version with a custom configuration (Fig. 1.4). There you should select value “C source files including models” for option “Code coverage details”. You should keep in mind that this will considerably increase the time necessary for generation of tasks and the overall time of the decision of the job version.



Decision: April 28, 2022, 6:05 p.m. (#1) **Author:** manager

#	Similar marks associations		Total verdict	Total status	Tags	Verifier			Klever version	Program fragmentation		Program fragment
	Confirmed	Automatic				CPU time	Wall time	Memory size		Tactic	Set	
1	0	0	Without marks	-	-	17 s	15 s	370 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /misc/ldusb.ko
2	0	0	Without marks	-	-	12 s	17 s	290 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /dwc3/dwc3-exynos.ko
3	0	0	Without marks	-	-	21 s	28 s	260 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /mon/usbmon.ko

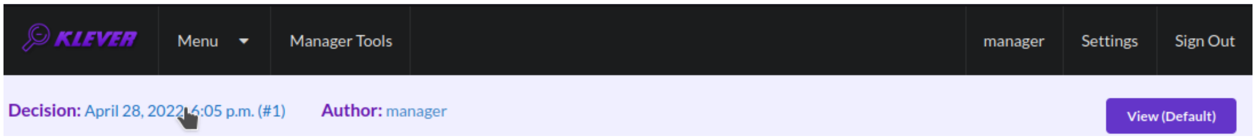
Fig. 1.19: The list of unsafes without any assessment



Decision: April 28, 2022, 6:05 p.m. (#1) **Author:** manager

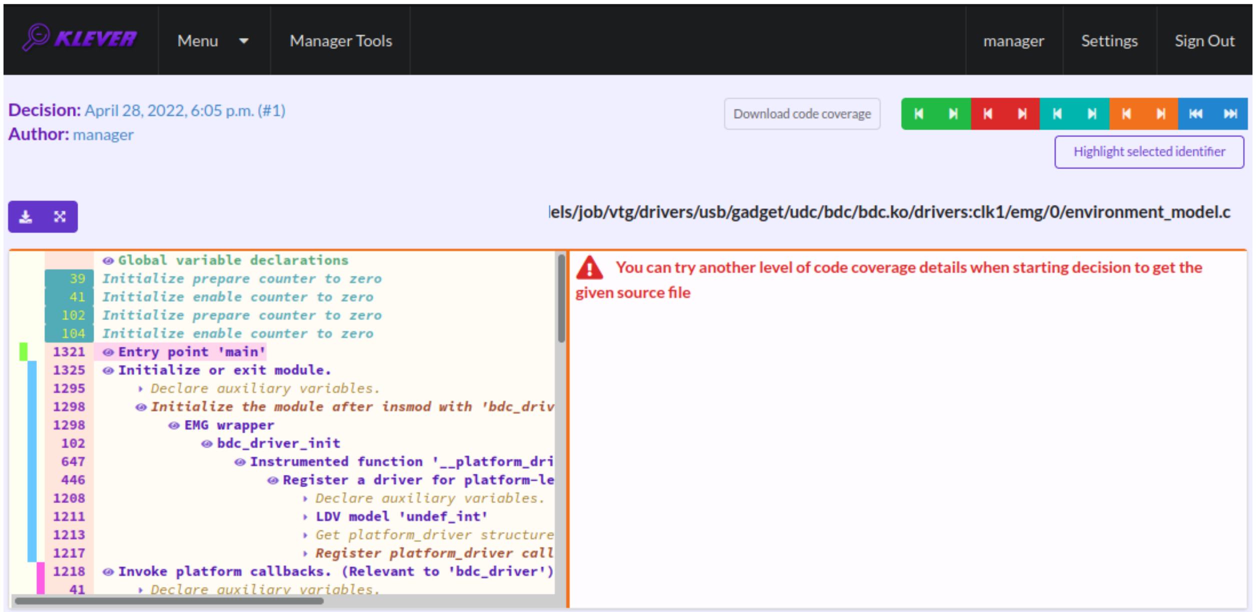
12	0	0	Without marks	-	-	23 s	31 s	250 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /phy/phy-tahvo.ko
13	0	0	Without marks	-	-	1.6 min	1.7 min	850 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /gadget/udc/bdc /bdc.ko
14	0	0	Without marks	-	-	57 s	1.1 min	1.7 GB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /gadget /udc/mv_udc.ko

Fig. 1.20: Opening the error trace corresponding to the unsafe without any assessment



Decision: April 28, 2022, 6:05 p.m. (#1) **Author:** manager

Fig. 1.21: Moving back to the job version/decision page



Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

lels/job/vtg/drivers/usb/gadget/udc/bdc/bdc.ko/drivers:clk1/emg/0/environment_model.c

```
     Global variable declarations
  39 Initialize prepare counter to zero
  41 Initialize enable counter to zero
 102 Initialize prepare counter to zero
 104 Initialize enable counter to zero
1321 Entry point 'main'
1325 Initialize or exit module.
1295     Declare auxiliary variables.
1298     Initialize the module after insmod with 'bdc_driv
1298         EMG wrapper
 102             bdc_driver_init
 647                 Instrumented function '__platform_dri
 446                     Register a driver for platform-le
1208                         Declare auxiliary variables.
1211                         LDV model 'undef_int'
1213                         Get platform_driver structure
1217                         Register platform_driver call
1218 Invoke platform callbacks. (Relevant to 'bdc_driver')
  41     Declare auxiliary variables.
```

You can try another level of code coverage details when starting decision to get the given source file

Fig. 1.22: The error trace for module drivers/usb/gadget/udc/bdc/bdc.ko and requirements specification drivers:clk1

as in Fig. 1.23. Error traces and source files are highlighted syntactically and you can use cross references for source files to find out definitions or places of usage for various entities.

You can click on eyes and on rectangles to show hidden parts of the error trace (Fig. 1.24-Fig. 1.25). Then you can hide them back if they are out of your interest. The difference between eyes and rectangles is that functions with eyes have either notes (Fig. 1.26) or warnings (Fig. 1.27) at some point of their execution, perhaps, within called functions. *Notes* describe important actions in models as well as those places in source files that are related to reported requirement violations from the standpoint of the verification tool. *Warnings* represent places where Klever detects violations of checked requirements.

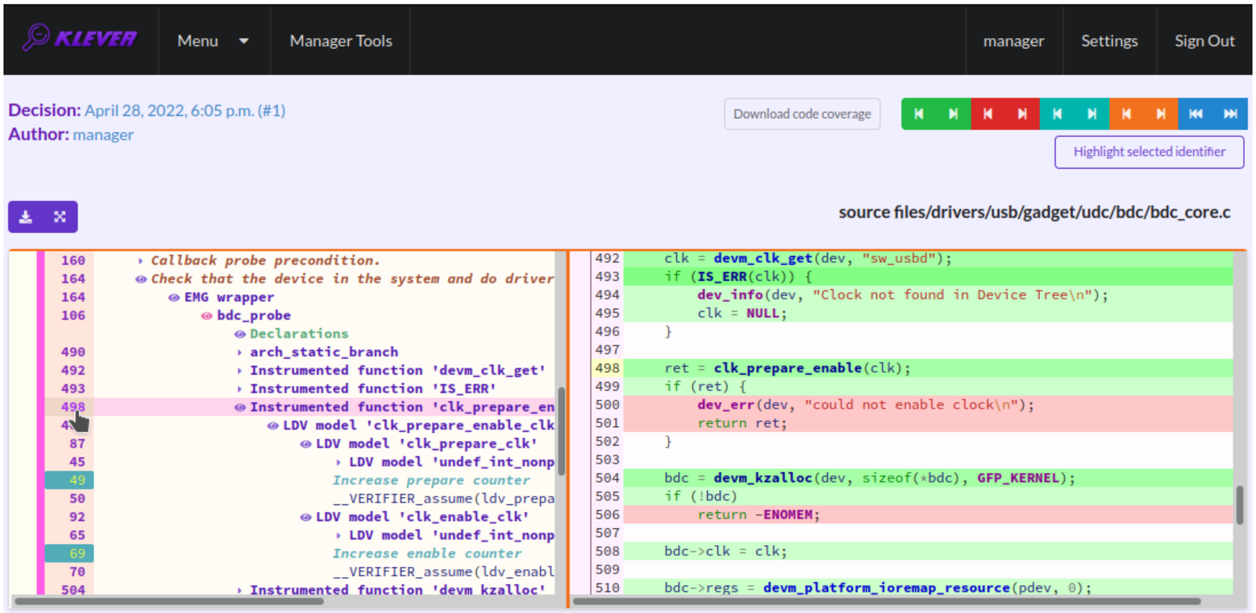
You can see that before calling module initialization and exit functions as well as module callbacks there is additional stuff in the error trace. These are parts of the environment model necessary to initialize models, to invoke module interfaces in the way the environment does and to check the final state. This tutorial does not consider models in detail, but you should keep in mind that Klever can detect faults not only directly in the source code under verification but also when checking something after execution of corresponding functions. For instance, this is the case for the considered error trace (Fig. 1.27).

1.2.7 Creating Marks

The analyzed unsafe corresponds to the fault that was fixed in upstream commits d2f42e09393c and 6f15a2a09cec to the Linux kernel. To finalize assessment you need to create a new *mark* (Fig. 1.28-Fig. 1.29):

1. Specify a verdict (**Bug** in our example).
2. Specify a status (**Fixed**).
3. Provide a description.
4. Save the mark.

After that you will be automatically redirected to the page demonstrating changes in total verdicts (Fig. 1.30). In our example there is the only change that corresponds to the analyzed unsafe and the new mark. But in a general case there may be many changes since the same mark can match several unsafes, and you may need to investigate these changes.



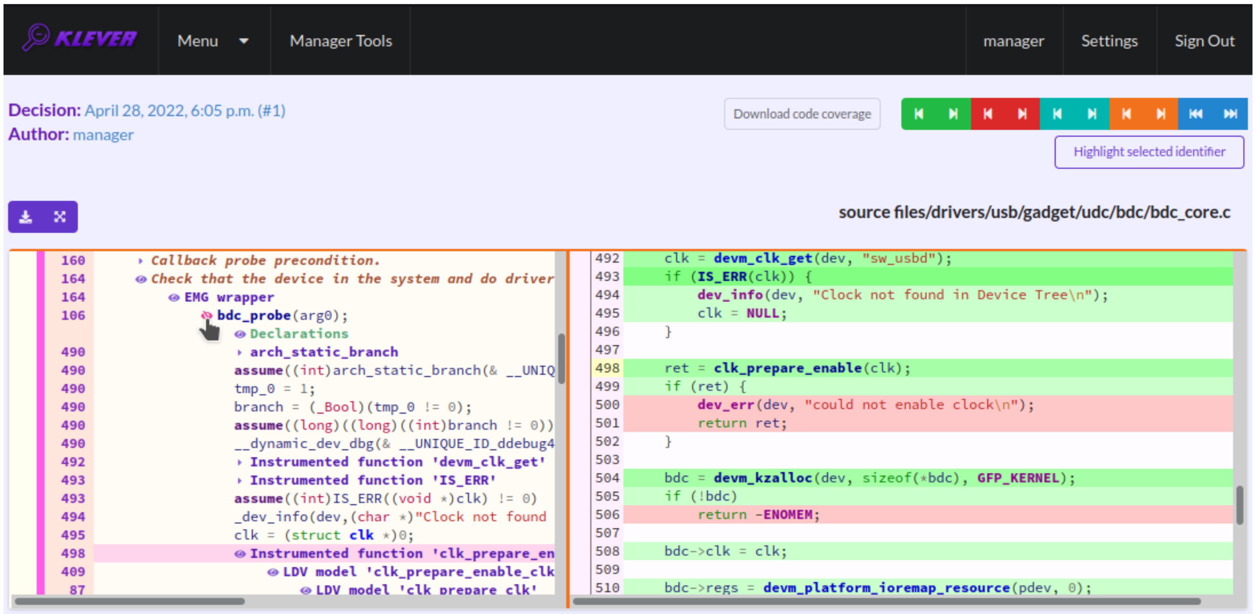
Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/udc/bdc/bdc_core.c

```
160      Callback probe precondition.
164     Check that the device in the system and do driver
164         EMG wrapper
106             bdc_probe
                    Declarations
490                 arch_static_branch
492                 Instrumented function 'devm_clk_get'
493                 Instrumented function 'IS_ERR'
498                 Instrumented function 'clk_prepare_en
4                       LDV model 'clk_prepare_enable_clk
87                          LDV model 'clk_prepare_clk'
45                              LDV model 'undef_int_nonp
49                              Increase prepare counter
50                              __VERIFIER_assume(ldv_prepa
92                          LDV model 'clk_enable_clk'
65                              LDV model 'undef_int_nonp
69                              Increase enable counter
70                              __VERIFIER_assume(ldv_enabl
504                 Instrumented function 'devm_kzalloc'
```

```
492     clk = devm_clk_get(dev, "sw_usbd");
493     if (IS_ERR(clk)) {
494         dev_info(dev, "Clock not found in Device Tree\n");
495         clk = NULL;
496     }
497
498     ret = clk_prepare_enable(clk);
499     if (ret) {
500         dev_err(dev, "could not enable clock\n");
501         return ret;
502     }
503
504     bdc = devm_kzalloc(dev, sizeof(*bdc), GFP_KERNEL);
505     if (!bdc)
506         return -ENOMEM;
507
508     bdc->clk = clk;
509
510     bdc->regs = devm_platform_ioremap_resource(pdev, 0);
```

Fig. 1.23: Showing the line in the original source file corresponding to the error trace statement



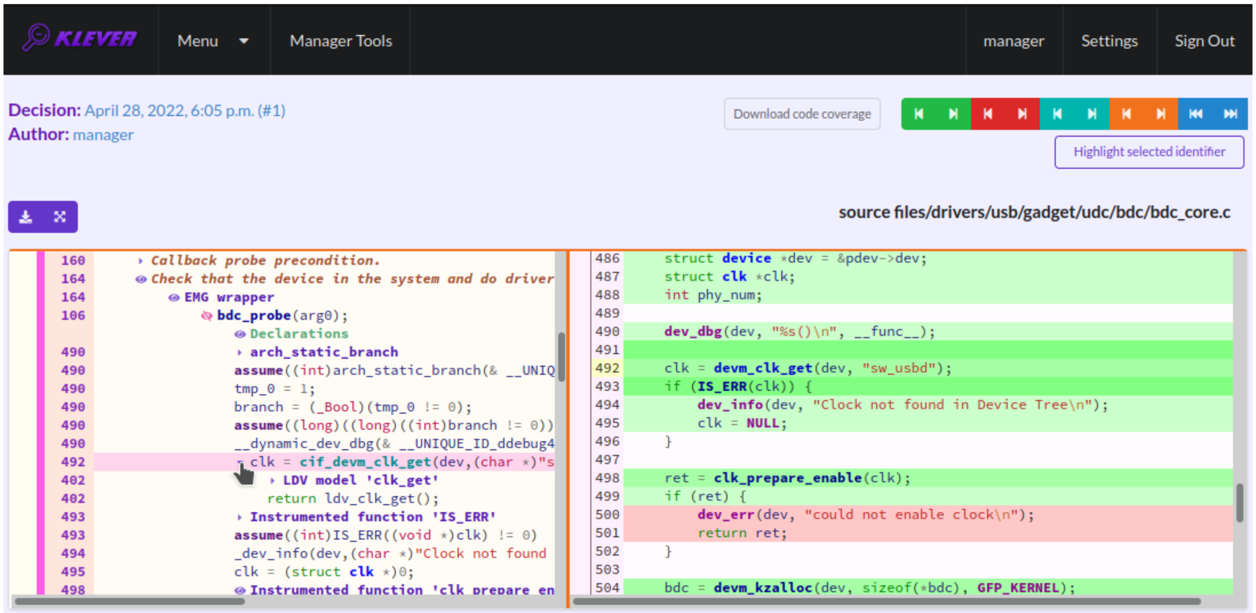
Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/udc/bdc/bdc_core.c

```
160      Callback probe precondition.
164     Check that the device in the system and do driver
164         EMG wrapper
106             bdc_probe(arg0);
                    Declarations
490                 arch_static_branch
490                 assume((int)arch_static_branch(& __UNIQ
490                 tmp_0 = 1;
490                 branch = (_Bool)(tmp_0 != 0);
490                 assume((long)((long)((int)branch != 0))
490                 __dynamic_dev_dbg(& __UNIQUE_ID_ddebug4
492                 Instrumented function 'devm_clk_get'
493                 Instrumented function 'IS_ERR'
493                 assume((int)IS_ERR((void *)clk) != 0)
494                 _dev_info(dev,(char *)"Clock not found
495                 clk = (struct clk *)0;
498                 Instrumented function 'clk_prepare_en
409                     LDV model 'clk_prepare_enable_clk
87                          LDV model 'clk_prepare_clk'
```

```
492     clk = devm_clk_get(dev, "sw_usbd");
493     if (IS_ERR(clk)) {
494         dev_info(dev, "Clock not found in Device Tree\n");
495         clk = NULL;
496     }
497
498     ret = clk_prepare_enable(clk);
499     if (ret) {
500         dev_err(dev, "could not enable clock\n");
501         return ret;
502     }
503
504     bdc = devm_kzalloc(dev, sizeof(*bdc), GFP_KERNEL);
505     if (!bdc)
506         return -ENOMEM;
507
508     bdc->clk = clk;
509
510     bdc->regs = devm_platform_ioremap_resource(pdev, 0);
```

Fig. 1.24: Showing hidden declarations, statements and assumptions for functions with notes or warnings



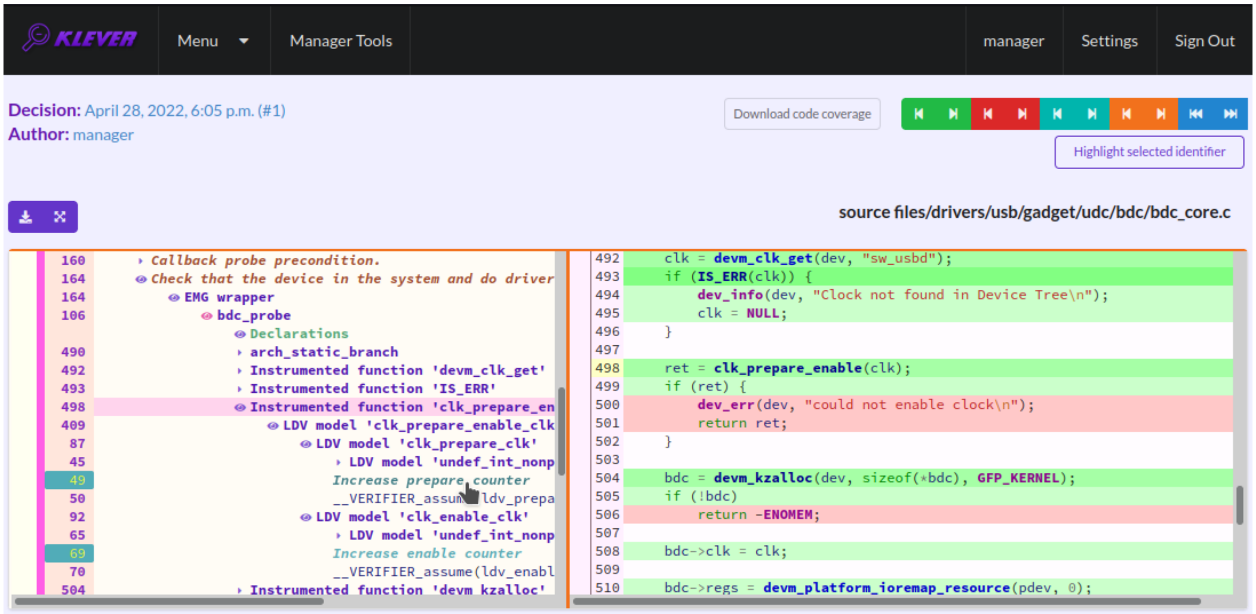
Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/udc/bdc/bdc_core.c

```
160       Callback probe precondition.
164      Check that the device in the system and do driver
164          EMG wrapper
106              bdc_probe(arg0);
                     Declarations
490                  arch_static_branch
490                  assume((int)arch_static_branch(& __UNIQ
490                  tmp_0 = 1;
490                  branch = (_Bool)(tmp_0 != 0);
490                  assume((long)((long)((int)branch != 0))
490                  __dynamic_dev_dbg(& __UNIQUE_ID_ddebug4
492                  clk = cif_devm_clk_get(dev,(char *)"s
402                      LDV model 'clk_get'
402                      return ldv_clk_get();
493                  Instrumented function 'IS_ERR'
493                  assume((int)IS_ERR((void *)clk) != 0)
494                  _dev_info(dev,(char *)"Clock not found
495                  clk = (struct clk *)0;
498                  Instrumented function 'clk_prepare_en
```

```
	struct device *dev = &pdev->dev;
	struct clk *clk;
	int phy_num;

	dev_dbg(dev, "%s()\n", __func__);

	clk = devm_clk_get(dev, "sw_usbd");
	if (IS_ERR(clk)) {
		dev_info(dev, "Clock not found in Device Tree\n");
		clk = NULL;
	}

	ret = clk_prepare_enable(clk);
	if (ret) {
		dev_err(dev, "could not enable clock\n");
		return ret;
	}

	bdc = devm_kzalloc(dev, sizeof(*bdc), GFP_KERNEL);
```

Fig. 1.25: Showing hidden declarations, statements and assumptions for functions without notes or warnings



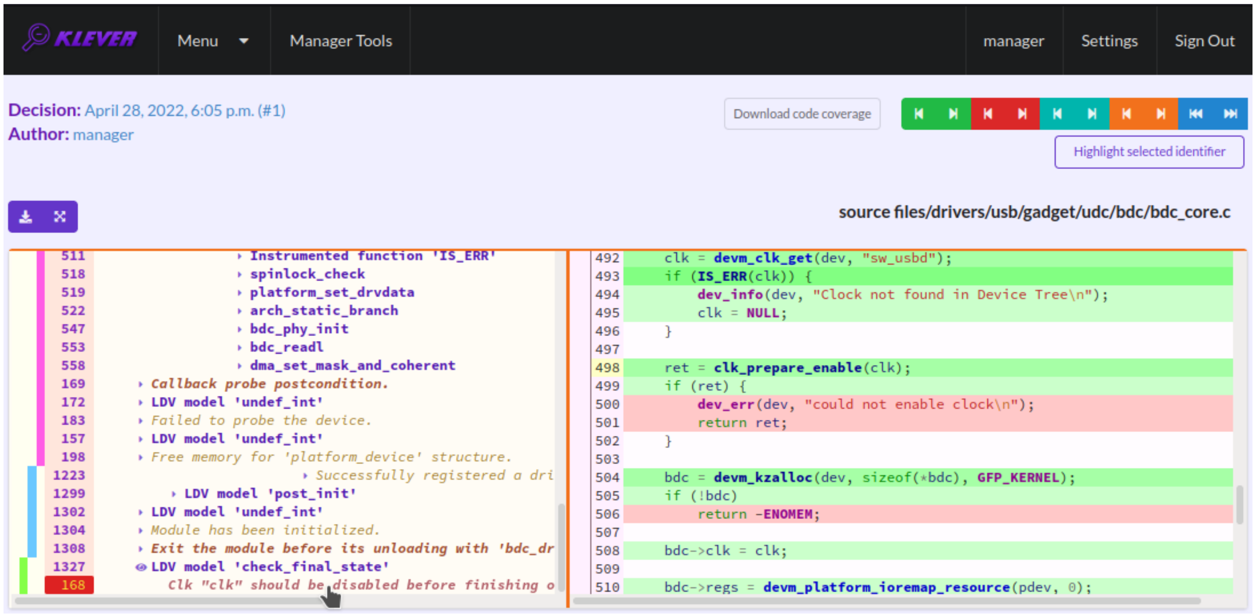
Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/udc/bdc/bdc_core.c

```
160       Callback probe precondition.
164      Check that the device in the system and do driver
164          EMG wrapper
106              bdc_probe
                     Declarations
490                  arch_static_branch
492                  Instrumented function 'devm_clk_get'
493                  Instrumented function 'IS_ERR'
498                  Instrumented function 'clk_prepare_en
409                      LDV model 'clk_prepare_enable_clk
 87                          LDV model 'clk_prepare_clk'
 45                              LDV model 'undef_int_nonp
 49                              Increase prepare counter
 50                              __VERIFIER_assum ldv_prepa
 92                          LDV model 'clk_enable_clk'
 65                              LDV model 'undef_int_nonp
 69                              Increase enable counter
 70                              __VERIFIER_assume(ldv_enabl
504                  Instrumented function 'devm_kzalloc'
```

```
	clk = devm_clk_get(dev, "sw_usbd");
	if (IS_ERR(clk)) {
		dev_info(dev, "Clock not found in Device Tree\n");
		clk = NULL;
	}

	ret = clk_prepare_enable(clk);
	if (ret) {
		dev_err(dev, "could not enable clock\n");
		return ret;
	}

	bdc = devm_kzalloc(dev, sizeof(*bdc), GFP_KERNEL);
	if (!bdc)
		return -ENOMEM;

	bdc->clk = clk;

	bdc->regs = devm_platform_ioremap_resource(pdev, 0);
```

Fig. 1.26: The error trace note



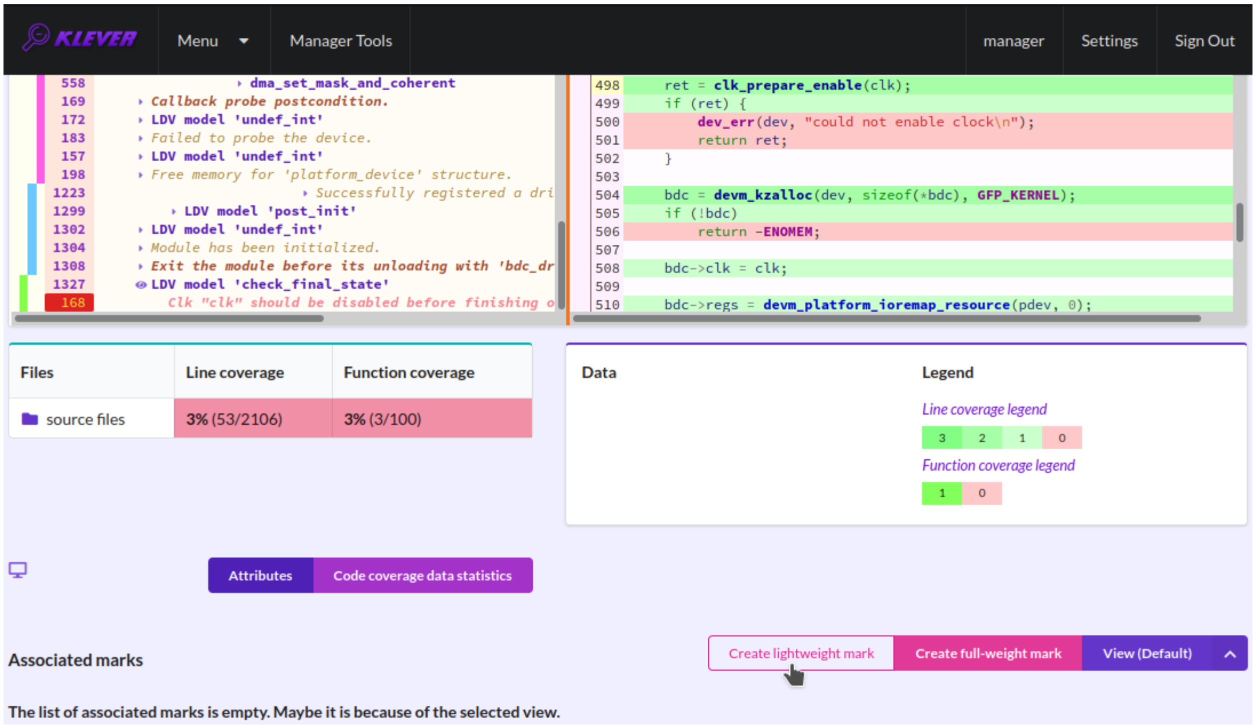
Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/udc/bdc/bdc_core.c

```
 511                  ‣ Instrumented function 'IS_ERR'
 518                  ‣ spinlock_check
 519                  ‣ platform_set_drvdata
 522                  ‣ arch_static_branch
 547                  ‣ bdc_phy_init
 553                  ‣ bdc_readl
 558                  ‣ dma_set_mask_and_coherent
 169      ‣ Callback probe postcondition.
 172      ‣ LDV model 'undef_int'
 183      ‣ Failed to probe the device.
 157      ‣ LDV model 'undef_int'
 198      ‣ Free memory for 'platform_device' structure.
1223                            ‣ Successfully registered a dri
1299          ‣ LDV model 'post_init'
1302      ‣ LDV model 'undef_int'
1304      ‣ Module has been initialized.
1308      ‣ Exit the module before its unloading with 'bdc_dr
1327      ◎ LDV model 'check_final_state'
 168          Clk "clk" should be disabled before finishing o
```

```
    clk = devm_clk_get(dev, "sw_usbd");
    if (IS_ERR(clk)) {
        dev_info(dev, "Clock not found in Device Tree\n");
        clk = NULL;
    }

    ret = clk_prepare_enable(clk);
    if (ret) {
        dev_err(dev, "could not enable clock\n");
        return ret;
    }

    bdc = devm_kzalloc(dev, sizeof(*bdc), GFP_KERNEL);
    if (!bdc)
        return -ENOMEM;

    bdc->clk = clk;

    bdc->regs = devm_platform_ioremap_resource(pdev, 0);
```

Fig. 1.27: The error trace warning



```
 558                  ‣ dma_set_mask_and_coherent
 169      ‣ Callback probe postcondition.
 172      ‣ LDV model 'undef_int'
 183      ‣ Failed to probe the device.
 157      ‣ LDV model 'undef_int'
 198      ‣ Free memory for 'platform_device' structure.
1223                            ‣ Successfully registered a dri
1299          ‣ LDV model 'post_init'
1302      ‣ LDV model 'undef_int'
1304      ‣ Module has been initialized.
1308      ‣ Exit the module before its unloading with 'bdc_dr
1327      ◎ LDV model 'check_final_state'
 168          Clk "clk" should be disabled before finishing o
```

```
    ret = clk_prepare_enable(clk);
    if (ret) {
        dev_err(dev, "could not enable clock\n");
        return ret;
    }

    bdc = devm_kzalloc(dev, sizeof(*bdc), GFP_KERNEL);
    if (!bdc)
        return -ENOMEM;

    bdc->clk = clk;

    bdc->regs = devm_platform_ioremap_resource(pdev, 0);
```

Files	Line coverage	Function coverage
source files	3% (53/2106)	3% (3/100)

Data

Legend

Line coverage legend

3	2	1	0

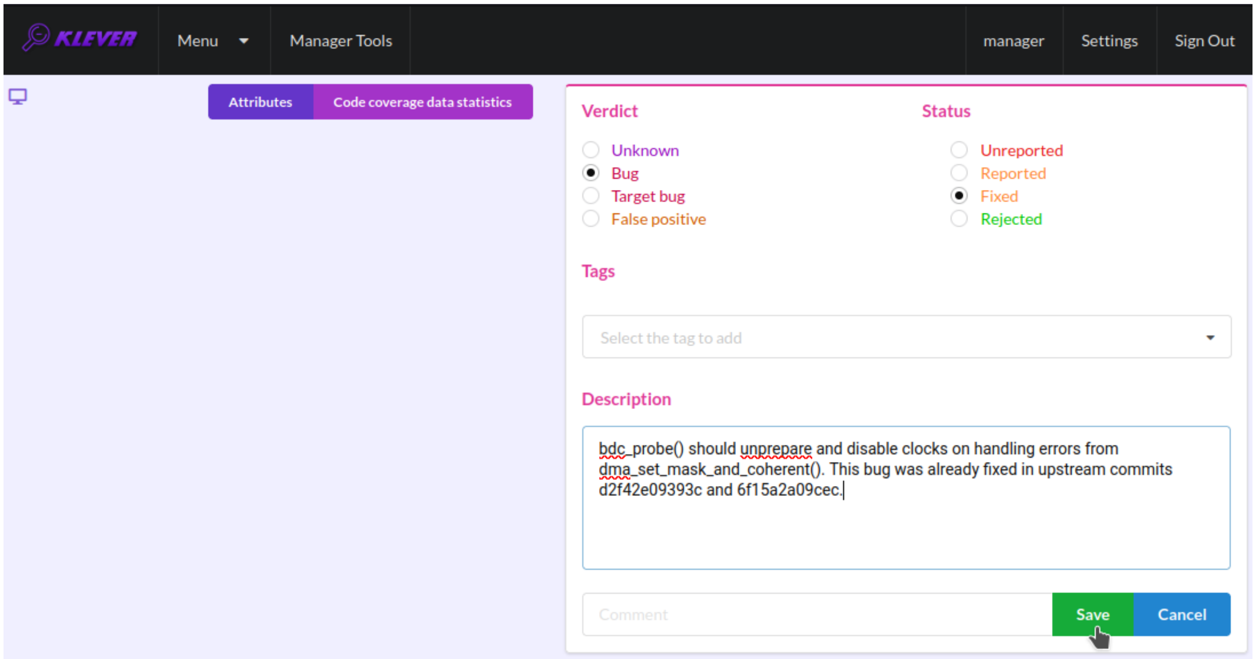
Function coverage legend

1	0

Associated marks

The list of associated marks is empty. Maybe it is because of the selected view.

Fig. 1.28: Starting the creation of a new lightweight mark



Verdict

○ Unknown
● Bug
○ Target bug
○ False positive

Status

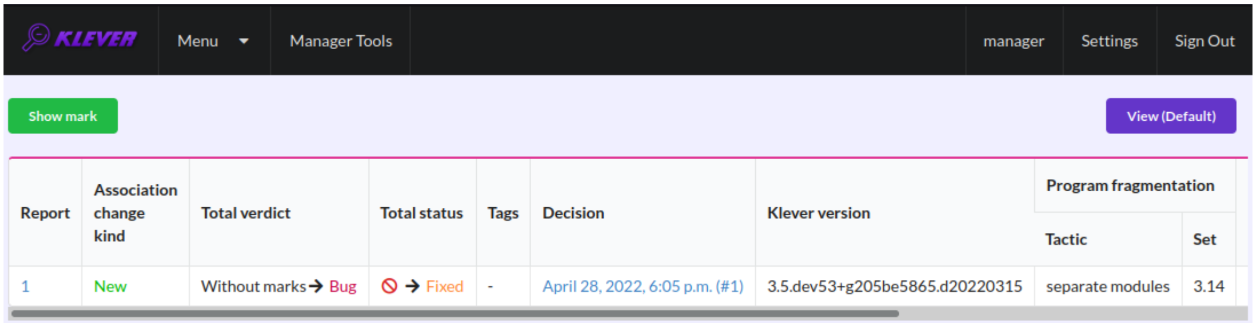
○ Unreported
○ Reported
● Fixed
○ Rejected

Tags

Description

bdc_probe() should unprepare and disable clocks on handling errors from dma_set_mask_and_coherent(). This bug was already fixed in upstream commits d2f42e09393c and 6f15a2a09cec.

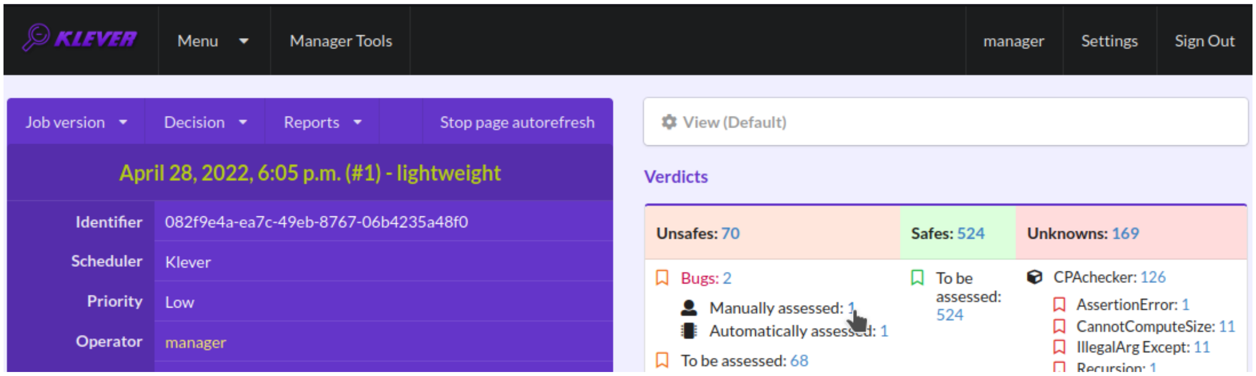
Fig. 1.29: The creation of the new lightweight mark



Report	Association change kind	Total verdict	Total status	Tags	Decision	Klever version	Program fragmentation	
							Tactic	Set
1	New	Without marks → Bug	⃠ → Fixed	-	April 28, 2022, 6:05 p.m. (#1)	3.5.dev53+g205be5865.d20220315	separate modules	3.14

Fig. 1.30: Changes in total verdicts

After creating the mark you can see the first manually assessed unsafe (Fig. 1.31). Besides, as it was already noted, you should investigate automatically assessed unsafes by analyzing corresponding error traces and marks and by (un)confirming their associations (Fig. 1.32-Fig. 1.33).



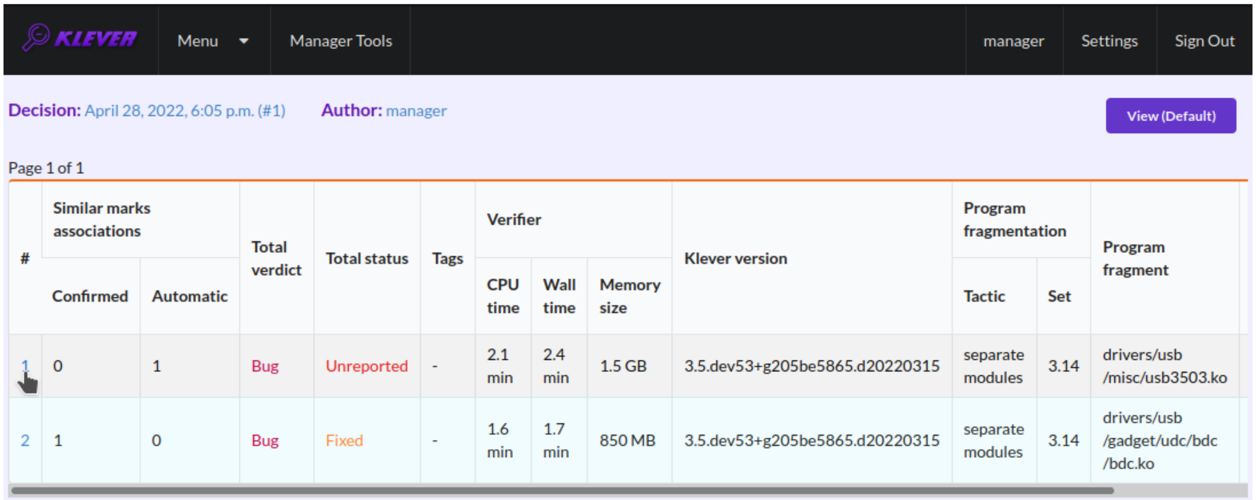
April 28, 2022, 6:05 p.m. (#1) - lightweight

Identifier	082f9e4a-ea7c-49eb-8767-06b4235a48f0
Scheduler	Klever
Priority	Low
Operator	manager

Verdicts

Unsafes: 70	Safes: 524	Unknowns: 169
Bugs: 2 Manually assessed: 1 Automatically assessed: 1 To be assessed: 68	To be assessed: 524	CPAchecker: 126 AssertionError: 1 CannotComputeSize: 11 IllegalArg Except: 11 Recursion: 1

Fig. 1.31: The total number of manually assessed unsafes



Decision: April 28, 2022, 6:05 p.m. (#1) Author: manager

Page 1 of 1

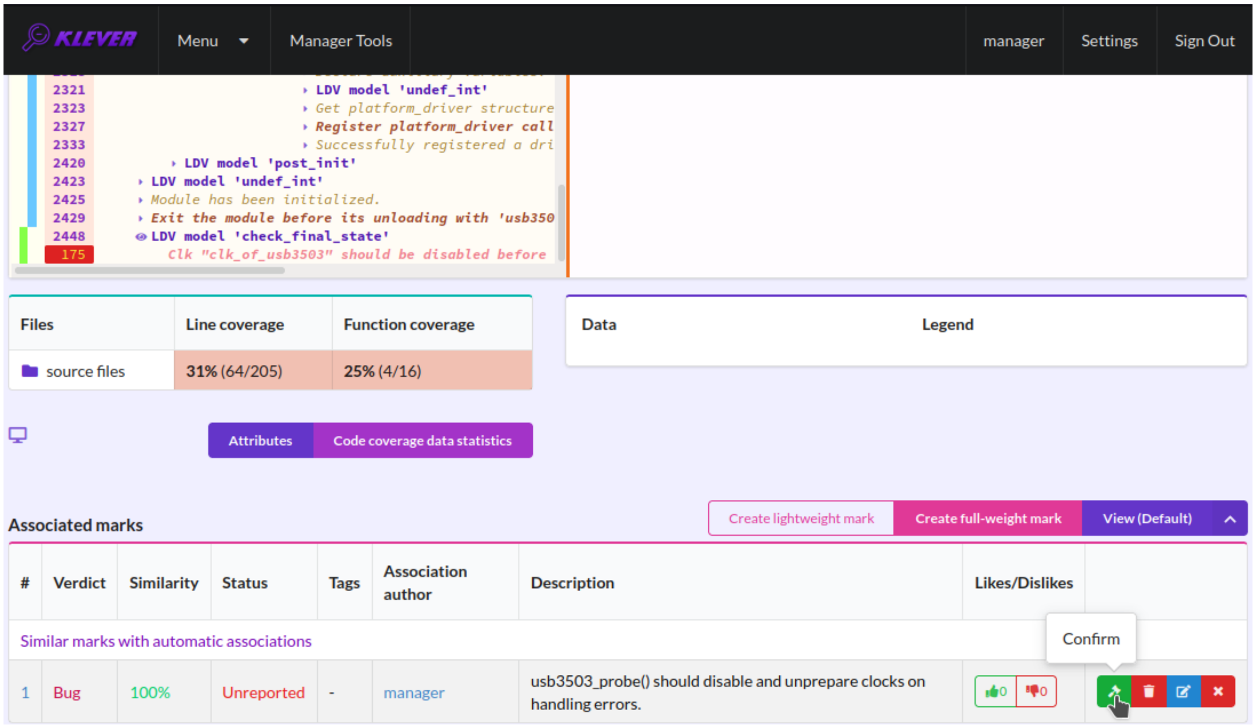
#	Similar marks associations		Total verdict	Total status	Tags	Verifier			Klever version	Program fragmentation		Program fragment
	Confirmed	Automatic				CPU time	Wall time	Memory size		Tactic	Set	
1	0	1	Bug	Unreported	-	2.1 min	2.4 min	1.5 GB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /misc/usb3503.ko
2	1	0	Bug	Fixed	-	1.6 min	1.7 min	850 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /gadget/udc/bdc /bdc.ko

Fig. 1.32: Opening the error trace of the unsafe with automatic assessment

False alarms can happen due to different reasons. You can find a tree of corresponding *tags* representing most common false alarm reasons at *Menu* → *Marks* → *Tags* (Fig. 1.34).

Each tag has a description that is shown when covering a tag name (Fig. 1.35).

You can choose appropriate tags during creation of marks from the dropdown list (Fig. 1.36). This list can be filtered out by entering parts of tag names (Fig. 1.37).



```
2321                    LDV model 'undef_int'
2323                    Get platform_driver structure
2327                    Register platform_driver call
2333                    Successfully registered a dri
2420        LDV model 'post_init'
2423    LDV model 'undef_int'
2425    Module has been initialized.
2429    Exit the module before its unloading with 'usb350
2448    LDV model 'check_final_state'
 175      Clk "clk_of_usb3503" should be disabled before
```

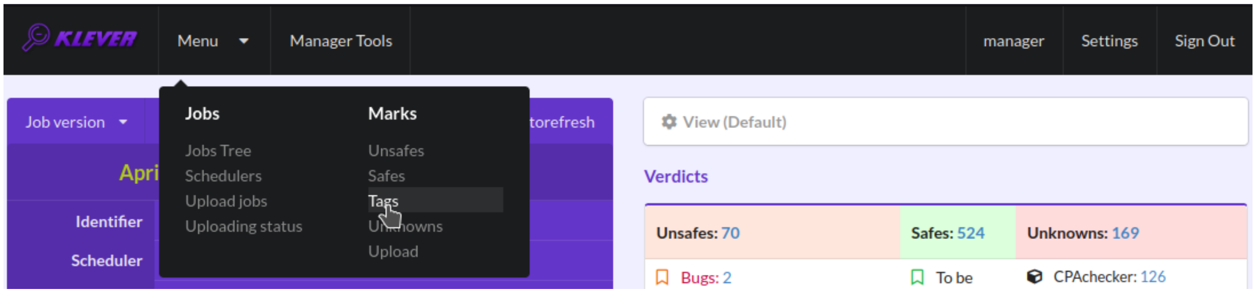
Files	Line coverage	Function coverage
source files	31% (64/205)	25% (4/16)

Data	Legend

Associated marks

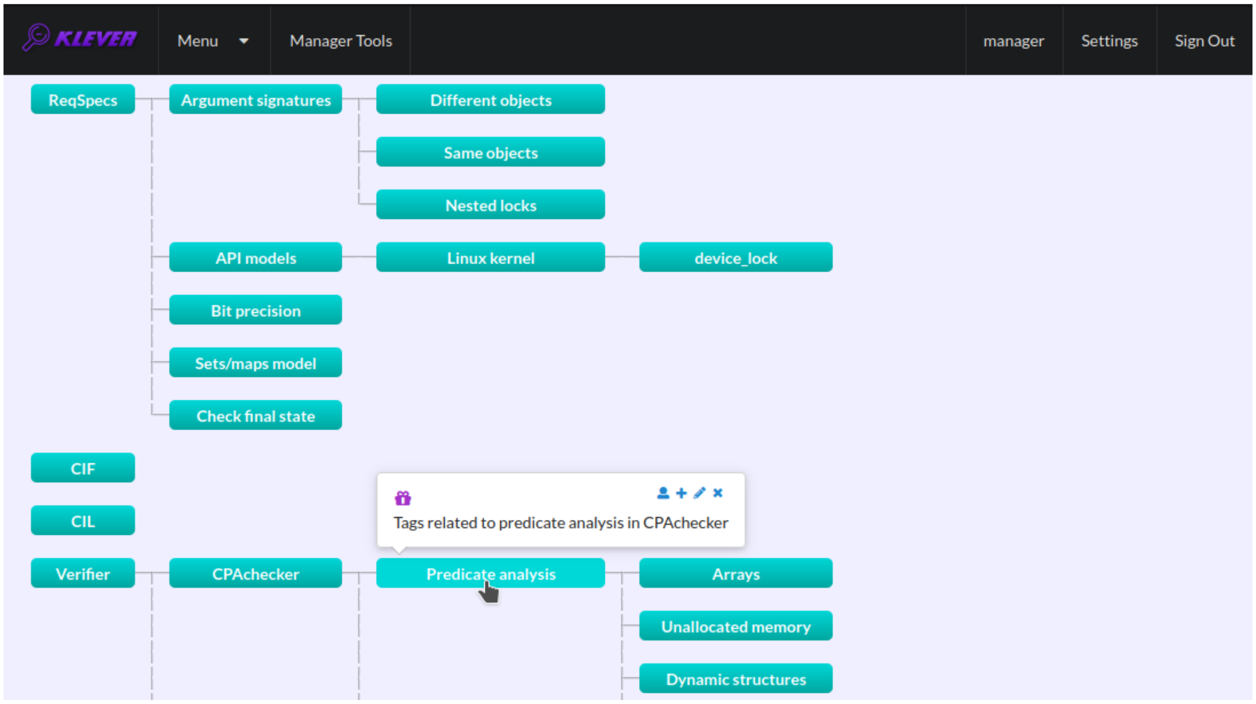
#	Verdict	Similarity	Status	Tags	Association author	Description	Likes/Dislikes
Similar marks with automatic associations							
1	Bug	100%	Unreported	-	manager	usb3503_probe() should disable and unprepare clocks on handling errors.	0 0

Fig. 1.33: Confirming the automatic association



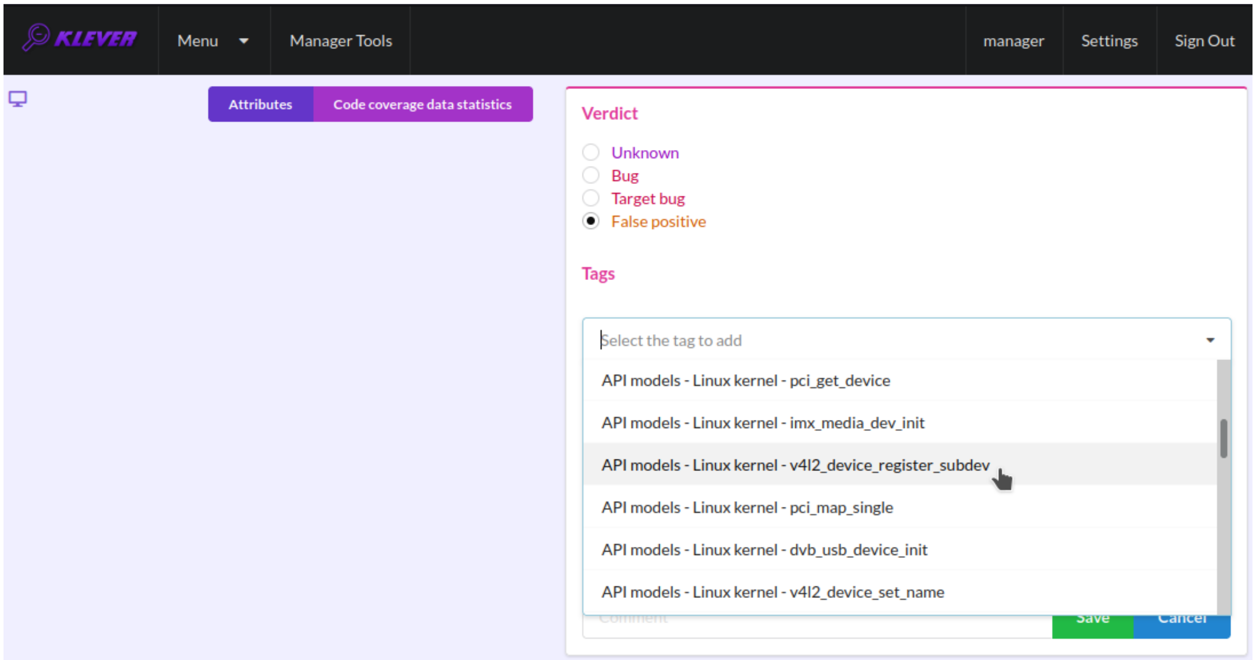
Unsafes: 70	Safes: 524	Unknowns: 169
Bugs: 2	To be	CPAchecker: 126

Fig. 1.34: Opening the tags page



Tags related to predicate analysis in CPAchecker

Fig. 1.35: Showing tag description

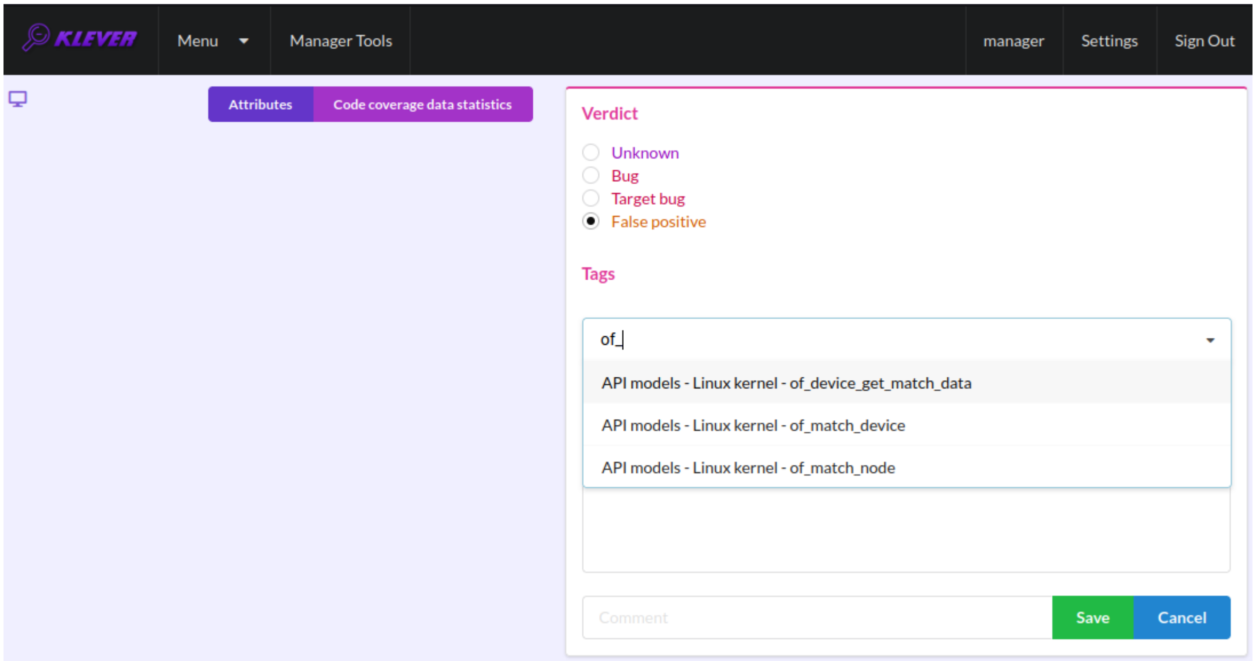


○ Unknown
○ Bug
○ Target bug
● False positive

Select the tag to add

- API models - Linux kernel - pci_get_device
- API models - Linux kernel - imx_media_dev_init
- API models - Linux kernel - v4l2_device_register_subdev
- API models - Linux kernel - pci_map_single
- API models - Linux kernel - dvb_usb_device_init
- API models - Linux kernel - v4l2_device_set_name

Fig. 1.36: Choosing tag from the dropdown list



Verdict

○ Unknown
○ Bug
○ Target bug
● False positive

Tags

of_

API models - Linux kernel - of_device_get_match_data

API models - Linux kernel - of_match_device

API models - Linux kernel - of_match_node

Fig. 1.37: Entering tag name part

1.2.8 Analysis of Code Coverage Reports

Code coverage reports demonstrate parts (lines and functions at the moment) of the target program source code and models that were considered during verification. Though users can expect complete code coverage because programs are analyzed statically, actually this may not be the case due to incomplete or inaccurate environment models that make some code unreachable or due to some limitations of verification tools, e.g. they can ignore calls of functions through function pointers. When users need good or excellent completeness of verification it is necessary to study code coverage reports.

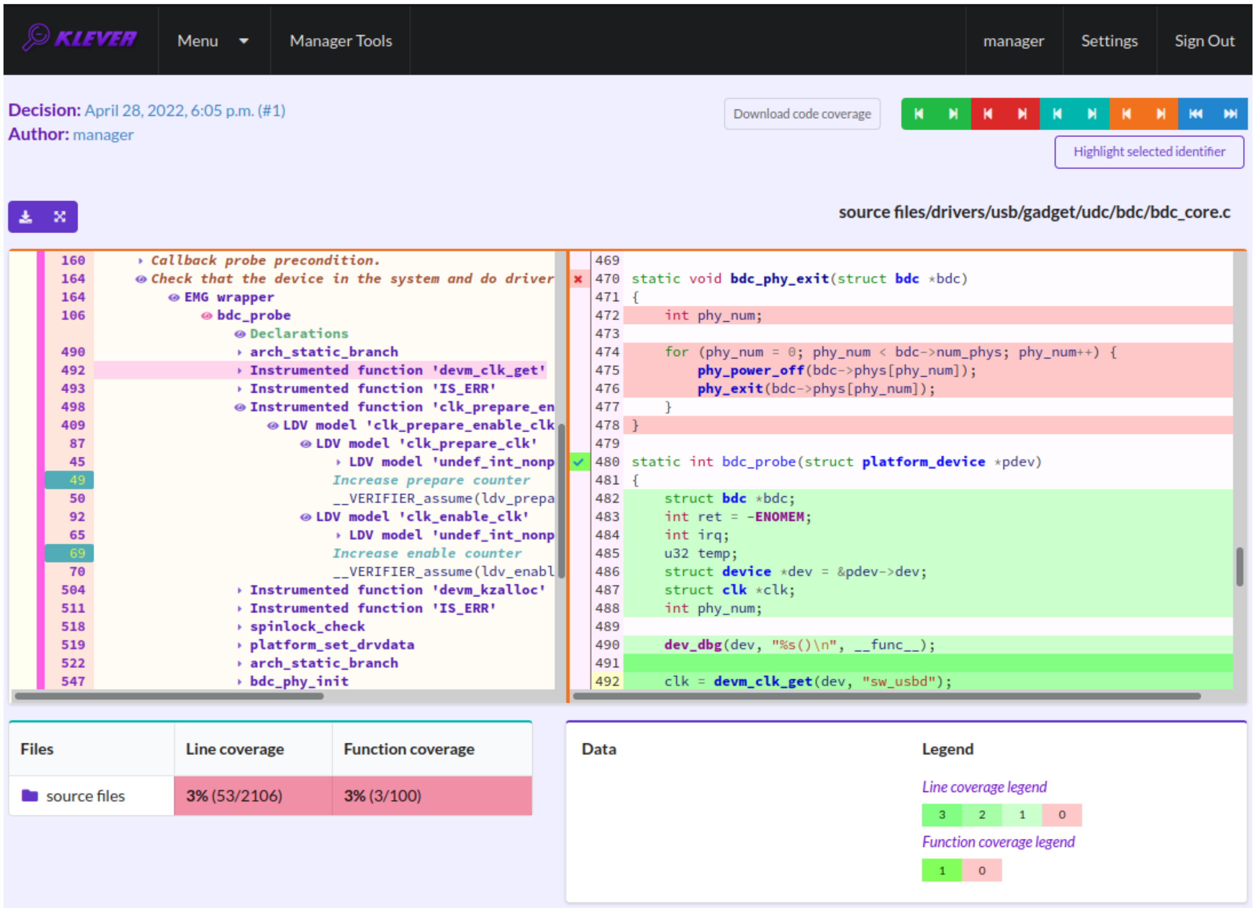
There is different semantics of code coverage for various verdicts:

- *Unsafes* - code coverage reports show exactly those parts of the source code that correspond to error traces. You will get another code coverage after eliminating reasons of corresponding unsafes.
- *Safes* - code coverage reports show all parts of the source code that the verification tool analyzed. You should keep in mind that there may be different reasons like specified above that prevent the verification tool from reaching complete code coverage. Since Klever lacks correctness proofs (currently, verification tools do not provide useful correctness proofs), analysis of code coverage reports becomes the only approach for understanding whether safes are good or not.
- *Unknowns* (*Timeouts*) - code coverage shows those parts of the target program source code that the verification tool could investigate until it was terminated after exhausting the specified amount of CPU time. You can find out and change corresponding limits in file *tasks.json* (for instance, see Fig. 1.4).

By default, Klever provides users with code coverage reports just for the target program source code. If one needs to inspect code coverage for various models it is necessary to start the decision of the job with a custom configuration where setting “Code coverage details” should be either “C source files including models” or “All source files”. This can result in quite considerable overhead, so it is not always switched on.

Code Coverage Reports for Unsafes

For unsafes, you will see code coverage reports when analyzing corresponding error traces like in Fig. 1.38. Code coverage of a particular source file is shown on the right. There is a code coverage legend beneath it. The pink background and red crosses point out uncovered lines and functions respectively. More times lines were analyzed during verification more intensive green background is used for them. Green ticks denote covered functions.



Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/udc/bdc/bdc_core.c

```
160    Callback probe precondition.
164    Check that the device in the system and do driver
164      EMG wrapper
106        bdc_probe
             Declarations
490          arch_static_branch
492          Instrumented function 'devm_clk_get'
493          Instrumented function 'IS_ERR'
498          Instrumented function 'clk_prepare_en
409            LDV model 'clk_prepare_enable_clk
87               LDV model 'clk_prepare_clk'
45                 LDV model 'undef_int_nonp
49                 Increase prepare counter
50                 __VERIFIER_assume(ldv_prepa
92               LDV model 'clk_enable_clk'
65                 LDV model 'undef_int_nonp
69                 Increase enable counter
70                 __VERIFIER_assume(ldv_enabl
504          Instrumented function 'devm_kzalloc'
511          Instrumented function 'IS_ERR'
518          spinlock_check
519          platform_set_drvdata
522          arch_static_branch
547          bdc_phy_init
```

```c

static void bdc_phy_exit(struct bdc *bdc)
{
    int phy_num;

    for (phy_num = 0; phy_num < bdc->num_phys; phy_num++) {
        phy_power_off(bdc->phys[phy_num]);
        phy_exit(bdc->phys[phy_num]);
    }
}

static int bdc_probe(struct platform_device *pdev)
{
    struct bdc *bdc;
    int ret = -ENOMEM;
    int irq;
    u32 temp;
    struct device *dev = &pdev->dev;
    struct clk *clk;
    int phy_num;

    dev_dbg(dev, "%s()\n", __func__);

    clk = devm_clk_get(dev, "sw_usbd");
```

Files	Line coverage	Function coverage
source files	3% (53/2106)	3% (3/100)

Data

Legend

Line coverage legend

3	2	1	0

Function coverage legend

1	0

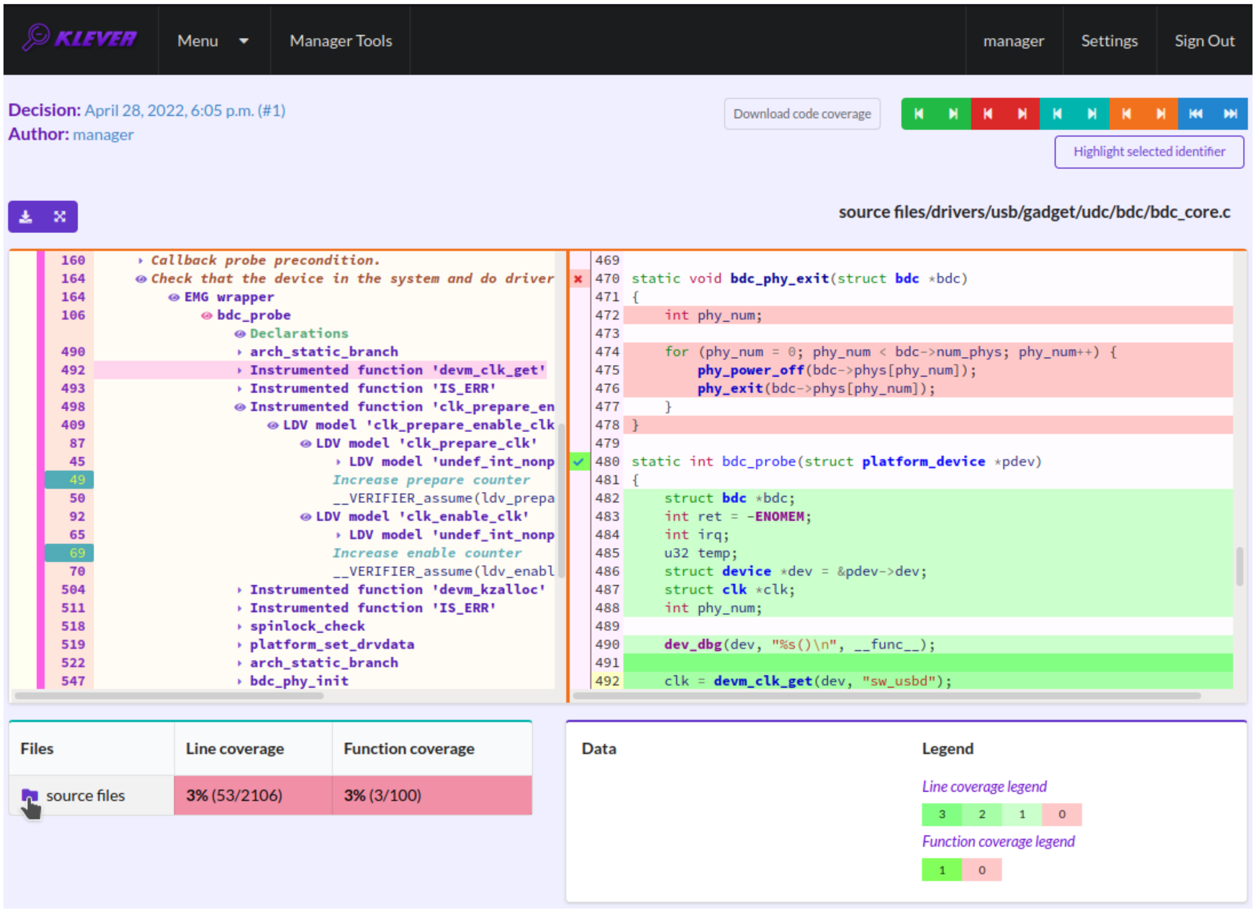
Fig. 1.38: Code coverage report for the unsafe error trace

There is code coverage statistics as well as a source tree on the left of the code coverage legend (Fig. 1.39). You can click on names of directories and source files to reveal corresponding statistics and to show code coverage for these source files (Fig. 1.40). The latter has sense for tasks consisting of several source files.

Code Coverage Reports for Safes

To open code coverage reports for safes you need to open a page with a list of safes (Fig. 1.41) and then open a particular safe page (Fig. 1.42). Like for unsafe you can analyze the code coverage legend and statistics as well as to show code coverage for particular source files (Fig. 1.43).

The safe verdict does not imply program correctness since some parts of the program could be not analyzed at all and thus uncovered. To navigate to the next uncovered function you should press the red button with the arrow (Fig. 1.44). Then you can find places where this uncovered function is invoked and why this was not done during verification (in the considered case this was due to lack of environment model specifications for callbacks of the *usb_class_driver* structure). Besides, while a function can be covered there may be uncovered lines within it. For instance, this may be the case due to the verification tool assumes that some conditions are always true or false.



Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/udc/bdc/bdc_core.c

```
160   › Callback probe precondition.
164   Check that the device in the system and do driver
164       EMG wrapper
106           bdc_probe
                  Declarations
490               › arch_static_branch
492               › Instrumented function 'devm_clk_get'
493               › Instrumented function 'IS_ERR'
498               Instrumented function 'clk_prepare_en
409                   LDV model 'clk_prepare_enable_clk
87                        LDV model 'clk_prepare_clk'
45                            › LDV model 'undef_int_nonp
49                            Increase prepare counter
50                            __VERIFIER_assume(ldv_prepa
92                        LDV model 'clk_enable_clk'
65                            › LDV model 'undef_int_nonp
69                            Increase enable counter
70                            __VERIFIER_assume(ldv_enabl
504               › Instrumented function 'devm_kzalloc'
511               › Instrumented function 'IS_ERR'
518               › spinlock_check
519               › platform_set_drvdata
522               › arch_static_branch
547               › bdc_phy_init
```

```

static void bdc_phy_exit(struct bdc *bdc)
{
    int phy_num;

    for (phy_num = 0; phy_num < bdc->num_phys; phy_num++) {
        phy_power_off(bdc->phys[phy_num]);
        phy_exit(bdc->phys[phy_num]);
    }
}

static int bdc_probe(struct platform_device *pdev)
{
    struct bdc *bdc;
    int ret = -ENOMEM;
    int irq;
    u32 temp;
    struct device *dev = &pdev->dev;
    struct clk *clk;
    int phy_num;

    dev_dbg(dev, "%s()\n", __func__);

    clk = devm_clk_get(dev, "sw_usbd");
```

Files	Line coverage	Function coverage
source files	**3%** (53/2106)	**3%** (3/100)

Data

Legend

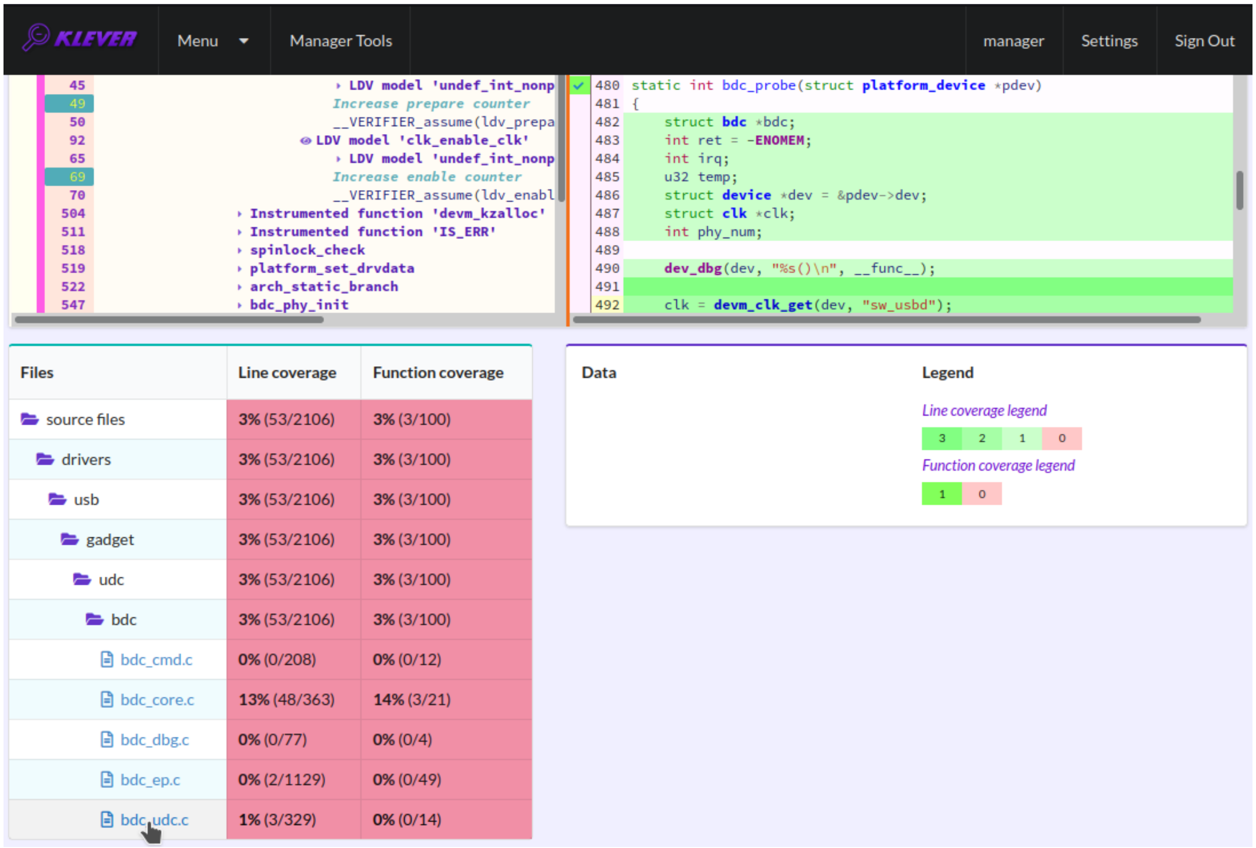
Line coverage legend

3	2	1	0

Function coverage legend

1	0

Fig. 1.39: Code coverage statistics



```
45                          LDV model 'undef_int_nonp
49                          Increase prepare counter
50                          __VERIFIER_assume(ldv_prepa
92                        LDV model 'clk_enable_clk'
65                          LDV model 'undef_int_nonp
69                          Increase enable counter
70                          __VERIFIER_assume(ldv_enabl
504               Instrumented function 'devm_kzalloc'
511               Instrumented function 'IS_ERR'
518               spinlock_check
519               platform_set_drvdata
522               arch_static_branch
547               bdc_phy_init
```

```
static int bdc_probe(struct platform_device *pdev)
{
	struct bdc *bdc;
	int ret = -ENOMEM;
	int irq;
	u32 temp;
	struct device *dev = &pdev->dev;
	struct clk *clk;
	int phy_num;

	dev_dbg(dev, "%s()\n", __func__);

	clk = devm_clk_get(dev, "sw_usbd");
```

Files	Line coverage	Function coverage
source files	3% (53/2106)	3% (3/100)
drivers	3% (53/2106)	3% (3/100)
usb	3% (53/2106)	3% (3/100)
gadget	3% (53/2106)	3% (3/100)
udc	3% (53/2106)	3% (3/100)
bdc	3% (53/2106)	3% (3/100)
bdc_cmd.c	0% (0/208)	0% (0/12)
bdc_core.c	13% (48/363)	14% (3/21)
bdc_dbg.c	0% (0/77)	0% (0/4)
bdc_ep.c	0% (2/1129)	0% (0/49)
bdc_udc.c	1% (3/329)	0% (0/14)

Data

Legend

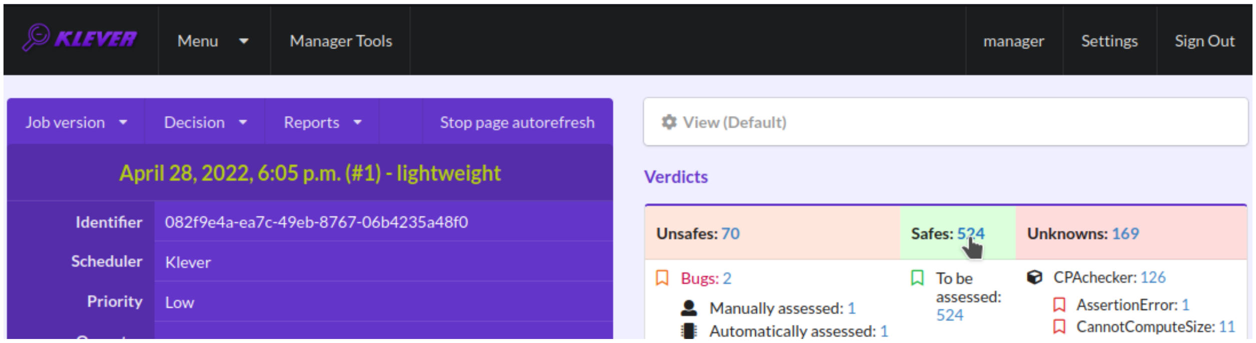
Line coverage legend

3 2 1 0

Function coverage legend

1 0

Fig. 1.40: Opening code coverage for the particular source file



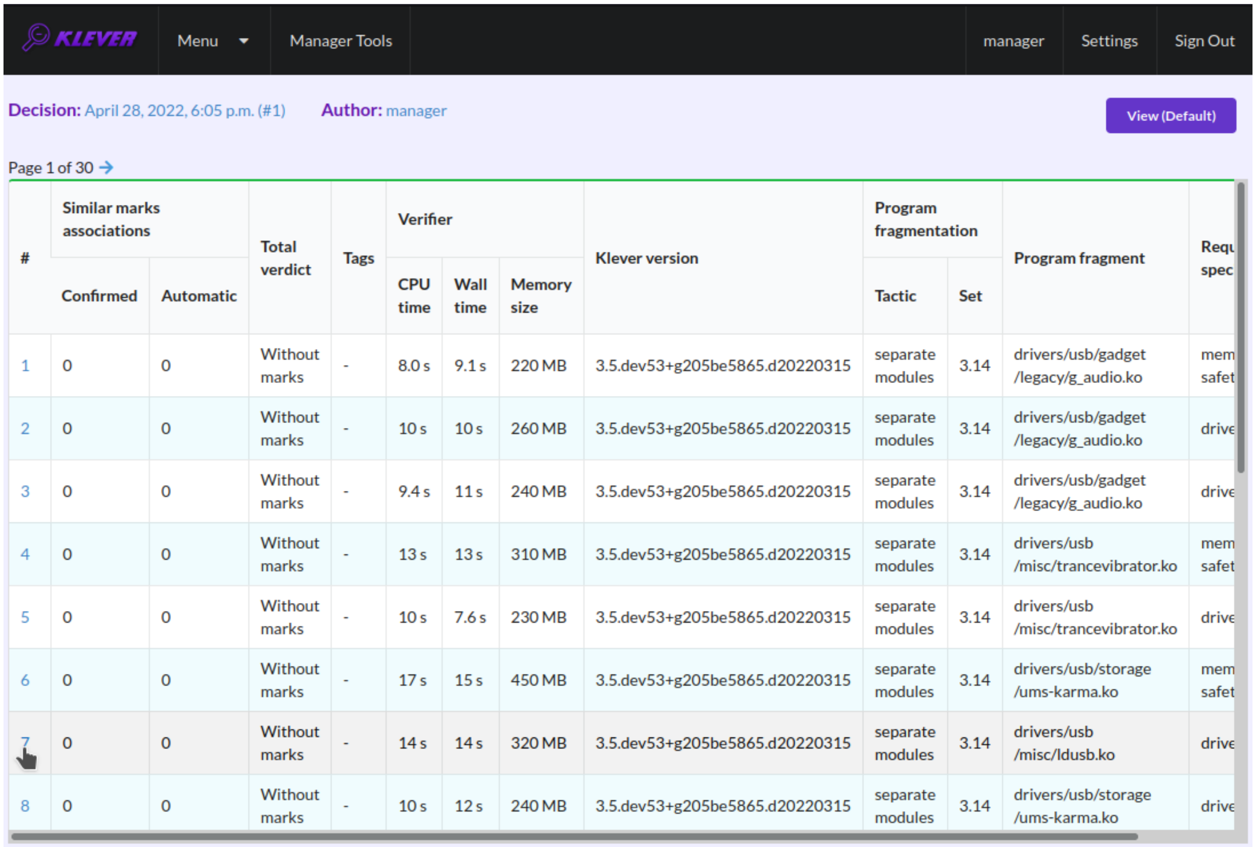
April 28, 2022, 6:05 p.m. (#1) - lightweight

Identifier	082f9e4a-ea7c-49eb-8767-06b4235a48f0
Scheduler	Klever
Priority	Low

Verdicts

Unsafes: 70	Safes: 524	Unknowns: 169
Bugs: 2 Manually assessed: 1 Automatically assessed: 1	To be assessed: 524	CPAchecker: 126 AssertionError: 1 CannotComputeSize: 11

Fig. 1.41: Opening page with the list of safes

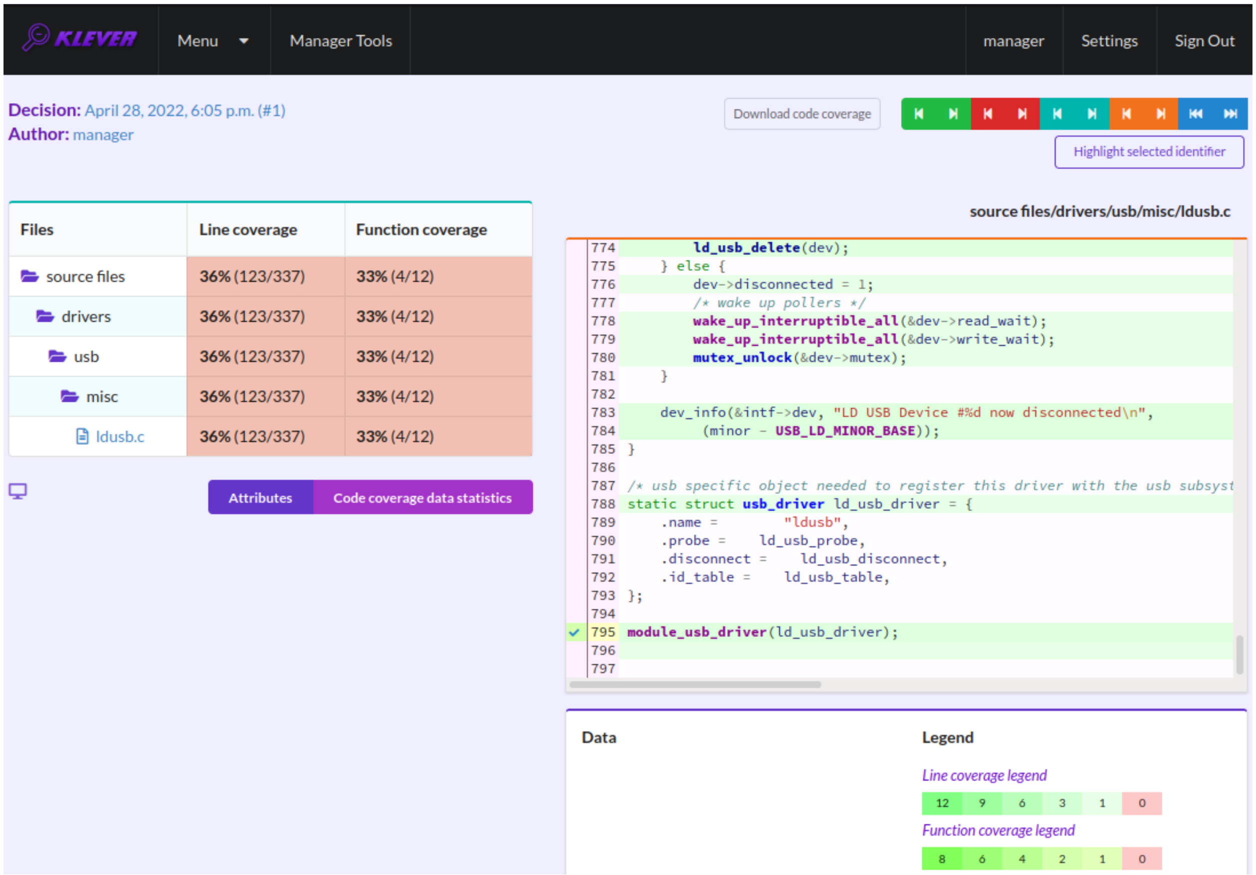


Decision: April 28, 2022, 6:05 p.m. (#1) **Author:** manager

Page 1 of 30 →

#	Similar marks associations		Total verdict	Tags	Verifier			Klever version	Program fragmentation		Program fragment	Requ spec
	Confirmed	Automatic			CPU time	Wall time	Memory size		Tactic	Set		
1	0	0	Without marks	-	8.0 s	9.1 s	220 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb/gadget /legacy/g_audio.ko	mem safet
2	0	0	Without marks	-	10 s	10 s	260 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb/gadget /legacy/g_audio.ko	drive
3	0	0	Without marks	-	9.4 s	11 s	240 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb/gadget /legacy/g_audio.ko	drive
4	0	0	Without marks	-	13 s	13 s	310 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /misc/trancevibrator.ko	mem safet
5	0	0	Without marks	-	10 s	7.6 s	230 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /misc/trancevibrator.ko	drive
6	0	0	Without marks	-	17 s	15 s	450 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb/storage /ums-karma.ko	mem safet
7	0	0	Without marks	-	14 s	14 s	320 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /misc/ldusb.ko	drive
8	0	0	Without marks	-	10 s	12 s	240 MB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb/storage /ums-karma.ko	drive

Fig. 1.42: Opening safe page



Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

Files	Line coverage	Function coverage
source files	36% (123/337)	33% (4/12)
drivers	36% (123/337)	33% (4/12)
usb	36% (123/337)	33% (4/12)
misc	36% (123/337)	33% (4/12)
ldusb.c	36% (123/337)	33% (4/12)

source files/drivers/usb/misc/ldusb.c

```
        ld_usb_delete(dev);
    } else {
        dev->disconnected = 1;
        /* wake up pollers */
        wake_up_interruptible_all(&dev->read_wait);
        wake_up_interruptible_all(&dev->write_wait);
        mutex_unlock(&dev->mutex);
    }

    dev_info(&intf->dev, "LD USB Device #%d now disconnected\n",
         (minor - USB_LD_MINOR_BASE));
}

/* usb specific object needed to register this driver with the usb subsyst
static struct usb_driver ld_usb_driver = {
    .name =        "ldusb",
    .probe =    ld_usb_probe,
    .disconnect =    ld_usb_disconnect,
    .id_table =    ld_usb_table,
};

module_usb_driver(ld_usb_driver);


```

Data

Legend

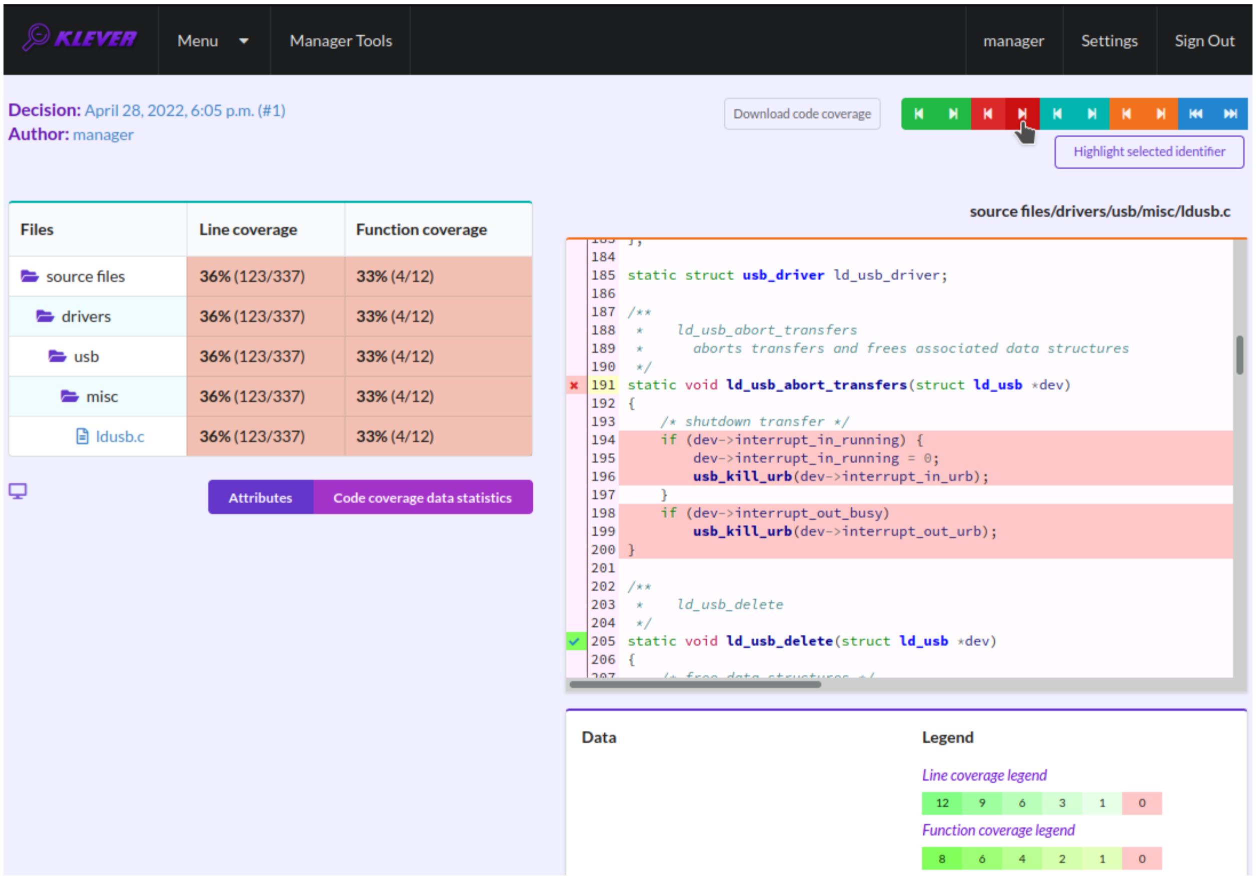
Line coverage legend

12	9	6	3	1	0

Function coverage legend

8	6	4	2	1	0

Fig. 1.43: Code coverage report for the safe



Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

Files	Line coverage	Function coverage
source files	36% (123/337)	33% (4/12)
drivers	36% (123/337)	33% (4/12)
usb	36% (123/337)	33% (4/12)
misc	36% (123/337)	33% (4/12)
ldusb.c	36% (123/337)	33% (4/12)

source files/drivers/usb/misc/ldusb.c

```
};

static struct usb_driver ld_usb_driver;

/**
 *    ld_usb_abort_transfers
 *      aborts transfers and frees associated data structures
 */
static void ld_usb_abort_transfers(struct ld_usb *dev)
{
    /* shutdown transfer */
    if (dev->interrupt_in_running) {
        dev->interrupt_in_running = 0;
        usb_kill_urb(dev->interrupt_in_urb);
    }
    if (dev->interrupt_out_busy)
        usb_kill_urb(dev->interrupt_out_urb);
}

/**
 *    ld_usb_delete
 */
static void ld_usb_delete(struct ld_usb *dev)
{
    /* free data structures */
```

Data

Legend

Line coverage legend

12	9	6	3	1	0

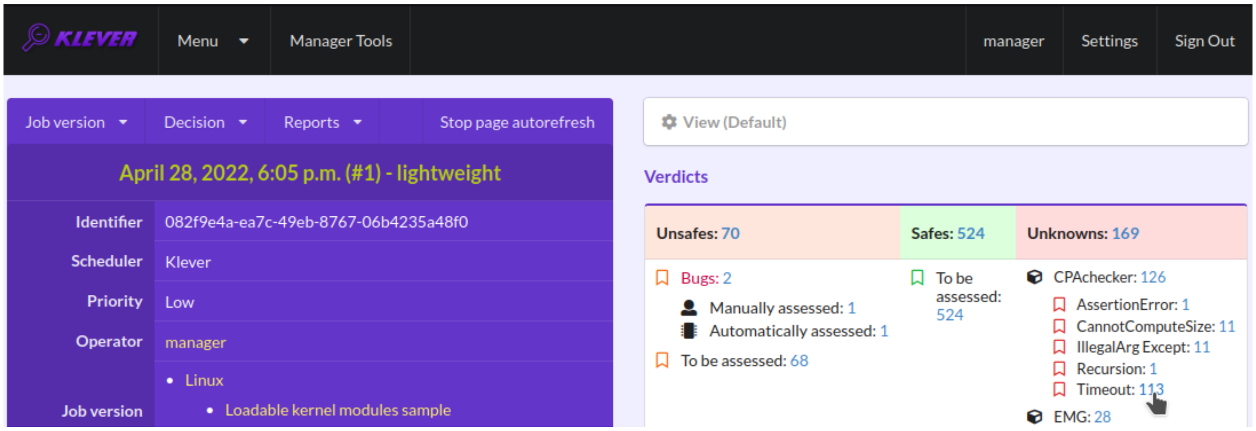
Function coverage legend

8	6	4	2	1	0

Fig. 1.44: Showing next uncovered function

Code Coverage Reports for Unknowns

If you would like to investigate the most complicated parts of the target program source code that can cause unknown (timeout) verdicts, you should open a page with a list of timeouts (Fig. 1.45) and then open a particular timeout page (Fig. 1.46). A timeout code coverage report (Fig. 1.47) looks almost like the safe code coverage report (Fig. 1.43).



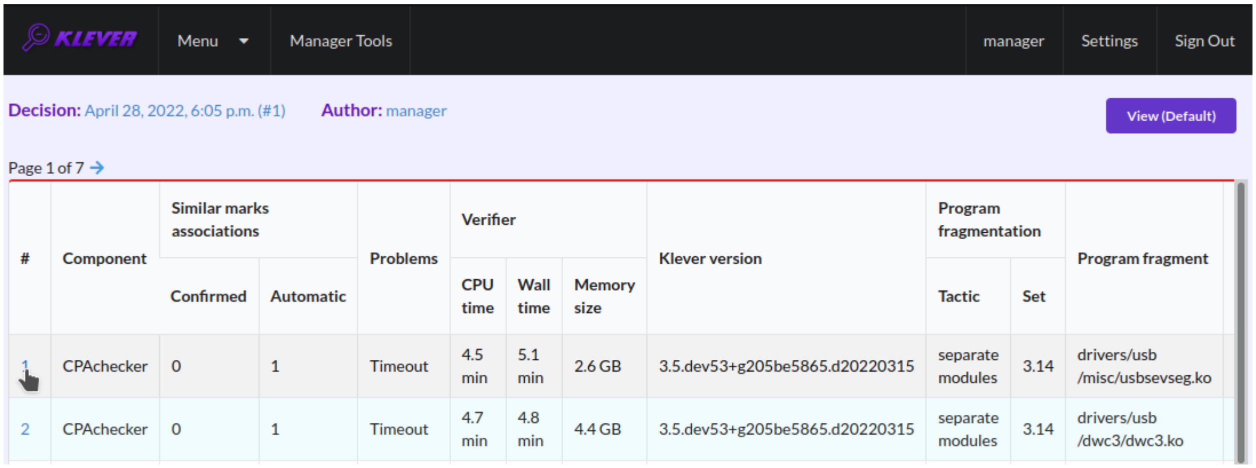
April 28, 2022, 6:05 p.m. (#1) - lightweight

Identifier	082f9e4a-ea7c-49eb-8767-06b4235a48f0
Scheduler	Klever
Priority	Low
Operator	manager
Job version	• Linux • Loadable kernel modules sample

Verdicts

Unsafes: 70	Safes: 524	Unknowns: 169
Bugs: 2 Manually assessed: 1 Automatically assessed: 1 To be assessed: 68	To be assessed: 524	CPAchecker: 126 AssertionError: 1 CannotComputeSize: 11 IllegalArg Except: 11 Recursion: 1 Timeout: 113 EMG: 28

Fig. 1.45: Opening page with the list of timeouts



Decision: April 28, 2022, 6:05 p.m. (#1) Author: manager

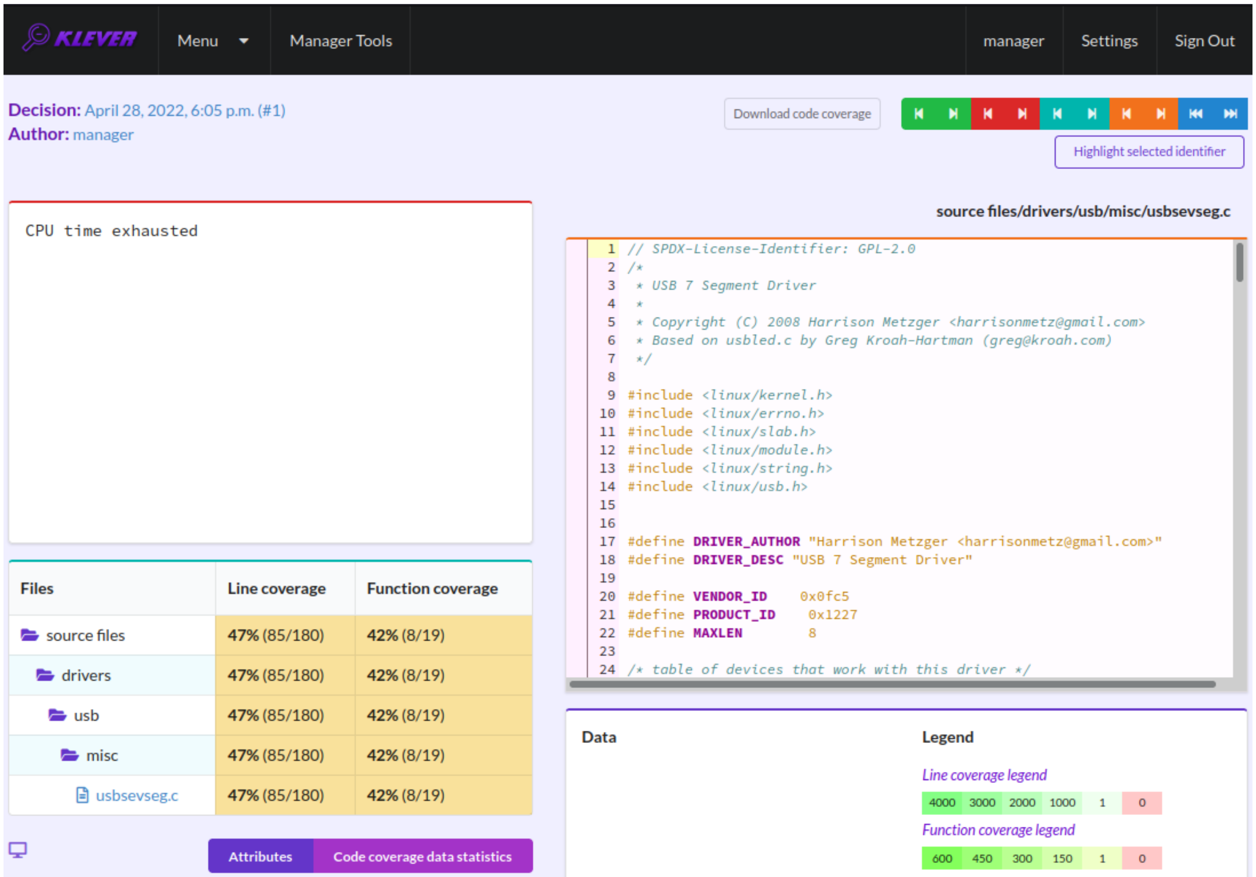
Page 1 of 7 →

#	Component	Similar marks associations		Problems	Verifier			Klever version	Program fragmentation		Program fragment
		Confirmed	Automatic		CPU time	Wall time	Memory size		Tactic	Set	
1	CPAchecker	0	1	Timeout	4.5 min	5.1 min	2.6 GB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /misc/usbsevseg.ko
2	CPAchecker	0	1	Timeout	4.7 min	4.8 min	4.4 GB	3.5.dev53+g205be5865.d20220315	separate modules	3.14	drivers/usb /dwc3/dwc3.ko

Fig. 1.46: Opening timeout page

To traverse through most covered lines that likely took most of the verification time you should press the orange button with the arrow (Fig. 1.48). If the task includes more than one source file it may be helpful for you to investigate lines that are most covered globally. For this it is necessary to press the blue button with the arrow. Quite often loops can serve as a source of complexity especially when loop boundaries are not specified/modelled explicitly.

You can find more details about verification results and their expert assessment in [G20].



Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

```
CPU time exhausted
```

Files	Line coverage	Function coverage
source files	47% (85/180)	42% (8/19)
drivers	47% (85/180)	42% (8/19)
usb	47% (85/180)	42% (8/19)
misc	47% (85/180)	42% (8/19)
usbsevseg.c	47% (85/180)	42% (8/19)

source files/drivers/usb/misc/usbsevseg.c

```
// SPDX-License-Identifier: GPL-2.0
/*
 * USB 7 Segment Driver
 *
 * Copyright (C) 2008 Harrison Metzger <harrisonmetz@gmail.com>
 * Based on usbled.c by Greg Kroah-Hartman (greg@kroah.com)
 */

#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/string.h>
#include <linux/usb.h>


#define DRIVER_AUTHOR "Harrison Metzger <harrisonmetz@gmail.com>"
#define DRIVER_DESC "USB 7 Segment Driver"

#define VENDOR_ID	0x0fc5
#define PRODUCT_ID	0x1227
#define MAXLEN		8

/* table of devices that work with this driver */
```

Data

Legend

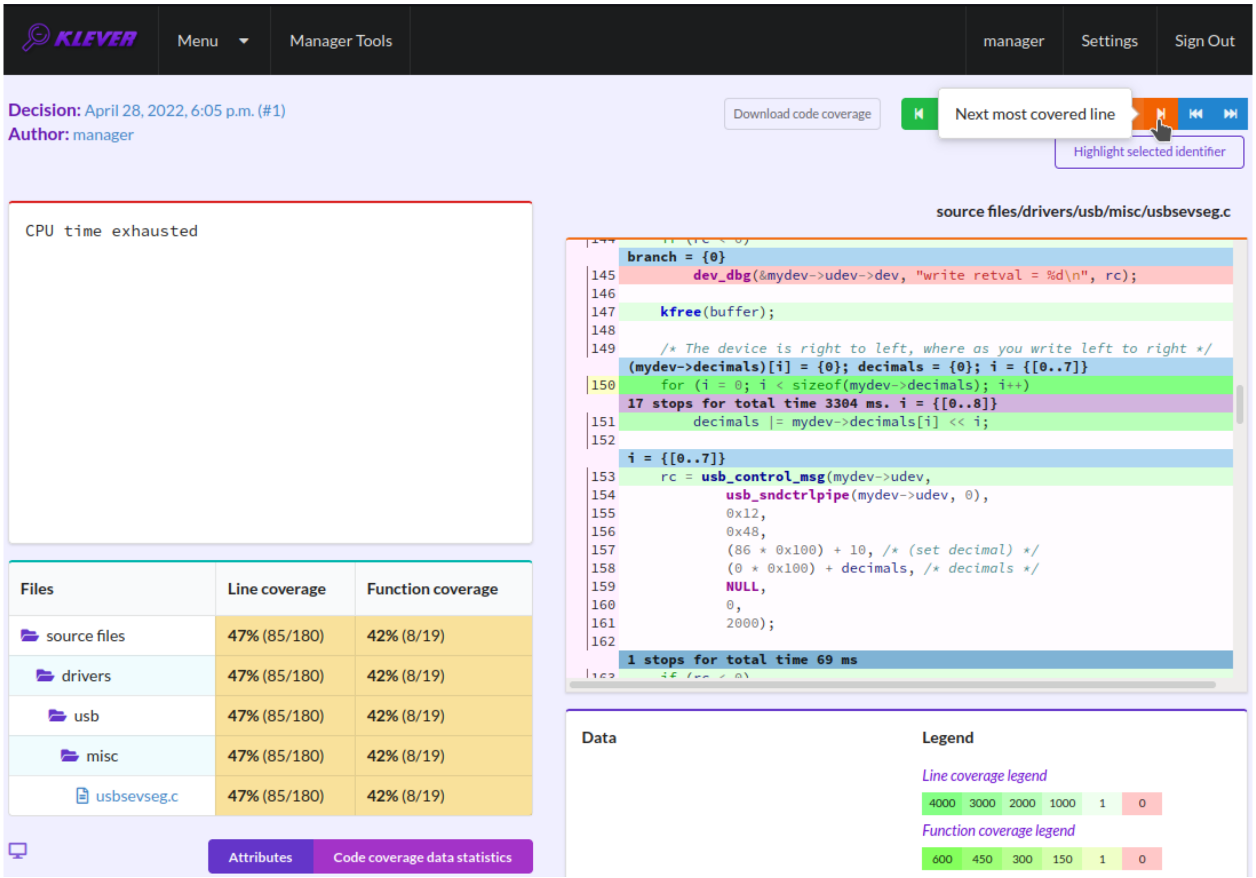
Line coverage legend

4000	3000	2000	1000	1	0

Function coverage legend

600	450	300	150	1	0

Fig. 1.47: Code coverage report for the timeout



Decision: April 28, 2022, 6:05 p.m. (#1)
Author: manager

```
CPU time exhausted
```

source files/drivers/usb/misc/usbsevseg.c

```
branch = {0}
        dev_dbg(&mydev->udev->dev, "write retval = %d\n", rc);

    kfree(buffer);

    /* The device is right to left, where as you write left to right */
(mydev->decimals)[i] = {0}; decimals = {0}; i = {[0..7]}
    for (i = 0; i < sizeof(mydev->decimals); i++)
17 stops for total time 3304 ms. i = {[0..8]}
        decimals |= mydev->decimals[i] << i;

i = {[0..7]}
    rc = usb_control_msg(mydev->udev,
            usb_sndctrlpipe(mydev->udev, 0),
            0x12,
            0x48,
            (86 * 0x100) + 10, /* (set decimal) */
            (0 * 0x100) + decimals, /* decimals */
            NULL,
            0,
            2000);

1 stops for total time 69 ms
```

Files	Line coverage	Function coverage
source files	47% (85/180)	42% (8/19)
drivers	47% (85/180)	42% (8/19)
usb	47% (85/180)	42% (8/19)
misc	47% (85/180)	42% (8/19)
usbsevseg.c	47% (85/180)	42% (8/19)

Data

Legend

Line coverage legend

4000	3000	2000	1000	1	0

Function coverage legend

600	450	300	150	1	0

Fig. 1.48: Showing next most covered line

1.2.9 What's Next?

We assume that you can be unsatisfied fully with a quality of obtained verification results. Perhaps, you even could not obtain them at all. This is expected since Klever is an open source software developed in Academy and we support verification of Linux kernel loadable modules for evaluation purposes primarily. Besides, this tutorial misses many various use cases of Klever. Some of these use cases are presented in other top-level sections of the user documentation. We are ready to discuss different issues and fix crucial bugs.

1.3 CLI

Klever supports a command-line interface for starting solution of verification jobs, for getting progress of their solution, etc. One can use CLI to automate usage of Klever, e.g. within CI. You should note that CLI is not intended for generation of *Klever Build Bases* and expert assessment of verification results.

This section describes several most important commands and the common workflow. In addition, it presents examples of using the corresponding Python API.

1.3.1 Setting Up Python API

Prior to refer to the Python API you need to set up an interface object. For default *Local Deployment* it can be done in the following way:

```
from klever.cli import Cli
cli = Cli(host=f'{hostname_or_ip}:8998', username='manager', password='manager')
```

You should specify these host and credentials as corresponding command-line arguments for all commands as well.

1.3.2 Starting Solution of Verification Jobs

You can start solution of a verification job based on any preset verification job. For this you should find out a corresponding identifier, **preset_job_id**, e.g. using Web UI. For instance, Linux loadable kernel modules sample has identifier “c1529fbf-a7db-4507-829e-55f846044309”. Then you should run something like:

```
$ klever-start-preset-solution --host $hostname_or_ip:8998 --username manager --password␣
↪manager $preset_job_id
```

In the output of this command there are:

- **job_id** - an identifier of the created verification job.
- **decision_id** - an identifier of a first version of the created verification job which decision was started.

There are several command-line arguments that you can use: *--rundata* and *--replacement*.

--rundata <job solution configuration file>

If you need some non-standard settings for solution of the verification job, e.g. you have a rather powerful machine and you want to use more parallel workers to generate verification tasks to speed up the complete process, you can provide a specific job solution configuration file. We recommend to develop an appropriate solution configuration using Web UI first and then you can download this file at the verification job page (e.g. *Decision* → *Download configuration*).

--replacement <JSON string or JSON file>

If you need to add some extra files in addition to files of the preset verification job or you want to replace some of them, you can describe corresponding changes using this option. For instance, you can provide a specific *Klever build base* and refer to it in **job.json**. In this case the value for this option may look like:

```
'{"job.json": "job.json", "loadable kernel modules sample.tar.gz": "loadable kernel modules sample.tar.gz"}'
```

File **job.json** and archive **loadable kernel modules sample.tar.gz** should be placed into the current working directory.

The corresponding Python API calls look as follows:

```
job_id = cli.create_job(preset_job_id)[1]
decision_id = cli.start_job_decision(job_id)[1]
```

For *start_job_decision* there are arguments *rundata* and *replacement* corresponding to `--rundata` and `--replacement`.

1.3.3 Waiting for Solution of Verification Job

Most likely you will need to wait for solution of the verification job whatever it will be successful or not. For this purpose you can execute something like:

```
$ klever-download-progress --host $hostname_or_ip:8998 --username manager --password manager -o progress.json $decision_id
```

until **status** in *progress.json* will be more than 2.

The appropriate invocation of the Python API may look like:

```
while True:
  time.sleep(5)
  progress = cli.decision_progress(decision_id)

  if int(progress['status']) > 2:
    break
```

1.3.4 Obtaining Verification Results

You can get verification results by using such the command:

```
$ klever-download-results --host $hostname_or_ip:8998 --username manager --password manager -o results.json $decision_id
```

or via the following Python API:

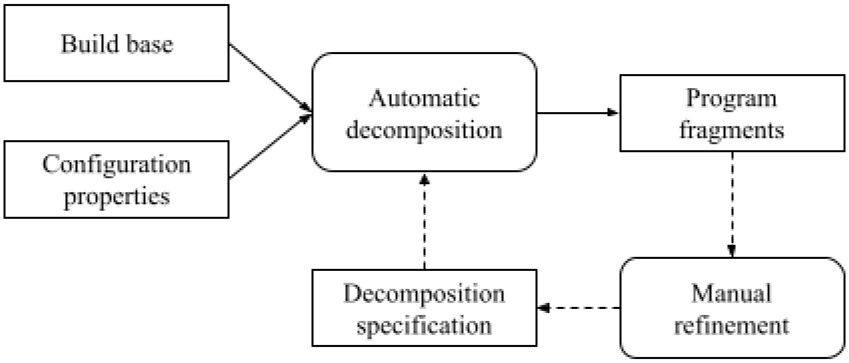
```
results = cli.decision_results(decision_id)
```

Then you can inspect file **results.json** or dictionary **results** somehow. Though, as it was noted, most likely you will need to analyze these results manually via Web UI.

1.4 Configuring Program Decomposition

The section presents a tutorial for program decomposition in Klever. Thorough verification is poorly scalable. Thus, one needs to bound the scope of each verification target. Decomposition of a program into *program fragments* is an essential solution for reducing verification complexity. The decomposition process should be done very accurately to balance required efforts for environment modeling and verification tool capabilities. It is hardly possible to propose any fully automatic solution to this problem. Klever provides means for both manual and automatic selection of source files for program fragments to meet user needs.

The Klever verification framework uses fragmentation tactics to decompose programs into fragments. Then composition tactics can combine program fragments into larger ones. There are additional decomposition specifications to control outputted program fragments' content. The figure below illustrates the workflow of Program Fragment Generator (PFG) that implements the approach in Klever.



We refer to each new type of programs to be verified as a project. Klever expects to have the fragmentation tactic implemented for the project in the `klever/core/pfg/fragmentation` directory of *$KLEVER_SRC*. You should use the *FragmentationAlgorythm* class as an ancestor to define your new fragmentation tactics. The new implementation may iterate over a build commands graph, program files graph and callgraph from the project's build base to fulfill the decomposition. Besides, there should be an appropriate decomposition specification in the `presets/jobs/fragmentation sets` directory of *$KLEVER_SRC*.

A new `job.json` file created for the project should have the following two options set there:

```
{
  "project": "project's name"
  "fragmentation tactic": "tactic's name"
}
```

Note that the configuration property *project* should match the tactic's file in the `klever/core/pfg/fragmentation` directory (the project name may have the capitalized wording). The second property helps to choose the fragmentation tactic when several variants are available. The provided value should be set in the file with the project's tactic name residing in the `presets/jobs/fragmentation sets` directory. Be default Klever uses the fragmentation tactic having attribute *reference* with the *true* value.

Let's consider the content of the `Linux.json` file to make it clear how decomposition specifications can be set up:

```
{
  "tactics": {
    "separate modules": {
      "reference": true,
```

(continues on next page)

(continued from previous page)

```
      "kernel": false,
      "ignore dependencies": true
    },
    "modules groups": {
      "kernel": false
    },
    "subsystems": {
      "kernel": true
    },
    "subsystems with modules": {
      "kernel": true,
      "add modules by coverage": true
    }
  },
  "fragmentation sets": {
    "3.14": {
      "reference": true,
      "fragments": {
        "drivers/usb/serial/usbserial.ko": [
          "drivers/usb/serial/usb_debug.ko",
          "drivers/usb/serial/usbserial.ko"
        ],
        "drivers/usb/serial/ch341.ko": [
          "drivers/usb/serial/ch341.ko",
          "drivers/usb/serial/generic.c"
        ],
        "drivers/usb/serial/usb_wwan.ko": [
          "drivers/usb/serial/usb_wwan.ko",
          "drivers/usb/serial/option.ko"
        ]
      }
    }
  }
}
```

The decomposition specification has two parts: *tactics* and *fragmentation sets*. The former contains set of configuration parameters for the fragmentation tactic that can be chosen by the *fragmentation tactic* configuration property in the `job.json` file. Options are specific to the tactic implementation. For instance, there are several variant in the aforementioned example:

- **separate modules** extracts loadable modules from the Linux kernel as individual program fragments.
- **module groups** works as the previous one but tries to merge some interdependent modules together heuristically in addition.
- **subsystems** extracts files from the same directory built in the main Linux kernel object file as separate program fragments called subsystems.
- **subsystems with modules** works as the previous one but tries to find modules that call functions exported by subsystems to check them together as new program fragments. This fragmentation tactic may require providing more input files, such as code coverage reports from previous Klever runs.

The second part of the decomposition specification allows to adjust program fragments manually. These program fragments are added to generated ones automatically or supersede program fragments with the same names. Entries of the attribute correspond to the project's versions that can be provided using the *specifications set* configuration property in the `job.json` file. Such sets can contain an enumeration of names of files, directories or existing program fragments for new program fragments given via the *fragments* attribute. Additional configuration properties *add to all fragments* and *exclude from all fragments* modify all generated program fragments. These options expect lists of file names with paths or regular expressions as values.

A user may choose any generated program fragments to verify by setting *targets* and *exclude targets* configuration properties in `job.json`. Program fragment names or regular expressions can be provided as values.

1.5 Development of Common API Models

Klever verifies *program fragments* rather than complete programs as a rule. Moreover, target programs can invoke library functions that are out of scope at verification. This can result in uncertain behavior of an environment. Software verification tools assume that invoked functions without definitions can return any possible value of their return types and do not have any side effects. Often users can agree with these implicit models especially taking into account that development of explicit models can take much time. For instance, this may be the case for functions that make debug printing and logging (at this stage we are not intended to check possible rules of usage of those APIs as well their implementations). Sometimes software verification tools can report false alarms or miss bugs, e.g. when invoked functions allocate memory, initialize it and return pointers to it. You should develop *common API models* if you are not satisfied with obtained verification results and if you have time for that. In addition to reducing obscure behavior you can leverage the same approach to decline a complexity of some internal APIs of considered program fragments. For instance, when the target program fragment contains a big loop that boundary depends on a value of an internal macro, you can try to decrease that value by developing an appropriate model.

Development of common API models is very similar to *Development of Requirement Specifications*. Here we will focus on some specific issues and tricks without repeating how to develop API models. You should enumerate additional common API models as a value of attribute **common models** of RSG plugin options within an appropriate requirement specifications base. It is necessary to keep in mind that common API models will be used for all requirement specifications unlike models developed for particular requirement specifications.

For functions without definitions you can omit aspect files since you can provide corresponding common API models as definitions of those functions. This way may be faster and easier but you should remember that one day your model can vanish suddenly due to function definitions will be considered as a part of target program fragments. For functions with definitions and for macros you have to develop aspect files anyway.

Regarding file names, we recommend following the same rules as for models for requirement specifications. In case of conflicts, i.e. when you need both common model and requirements specification model for the same API and, thus, the same file name, you should use suffix **.common.c** for the former. In case of such conflicts you can also have coinciding names of model functions, say, when you need to develop a model for a given function and check for its usage simultaneously. Moreover, this may be the case due to models for some functions, e.g. registration and deregistration ones, are defined within the generated environment model already. You have to define models with unique names and relate them with each other in such the way that will not prevent their original intention.

The last but not the least advice:

- Look at existing common API models. They can help you to learn the specific syntax as well as to investigate some particular working decisions.
- You should accurately model possible error behavior of modeled APIs. Otherwise, corresponding error handling paths will not be considered at verification that can lead to missing bugs.
- Do not forget to test your common API models like requirement specifications.

1.5.1 Example of Common API Model

Let's consider an example of development of a common API model. In the Linux kernel there is function *kzalloc()*. This is a vital function since a lot of loadable kernel modules use it and it affects subsequent execution paths very considerably. Moreover, it is necessary to check that callers invoke this function in the atomic context when passing **GFP_ATOMIC** as a value of argument **flags**.

static inline void *`kzalloc`(size_t size, gfp_t flags)

Allocate memory and initialize it with zeroes.

Parameters

- `size` – The size of memory to be allocated.
- `flags` – The type of memory to be allocated.

Returns

The pointer to the allocated and initialized memory in case of success and NULL otherwise.

The *kzalloc()* model can look as follows:

```
#include <linux/types.h>
#include <ldv/linux/common.h>
#include <ldv/linux/slab.h>
#include <ldv/verifier/memory.h>

void *ldv_kzalloc(size_t size, gfp_t flags)
{
    void *res;

    ldv_check_alloc_flags(flags);
    res = ldv_zalloc(size);

    return res;
}
```

Above we included several headers in the model:

- `ldv/linux/common.h` holds a declaration for **ldv_check_alloc_flags()**. Its definition may be provided by appropriate requirement specifications.
- `ldv/linux/slab.h` contains a declaration for a model function itself. Its possible content is demonstrated below.
- `ldv/verifier/memory.h` describes a bunch of memory allocation function models. In particular, **ldv_zalloc()** behaves exactly as *kzalloc()* without paying any attention to **flags**.

```
#ifndef __LDV_LINUX_SLAB_H
#define __LDV_LINUX_SLAB_H

#include <linux/types.h>

extern void *ldv_kzalloc(size_t size, gfp_t flags);

#endif /* __LDV_LINUX_SLAB_H */
```

We have to develop the aspect file since *kzalloc()* is a static inline function, i.e. it will have the definition always. The aspect file may be so:

```
before: file("$this")
{
#include <ldv/linux/slab.h>
}


around: execution(static inline void *kzalloc(size_t size, gfp_t flags))
{
    return ldv_kzalloc(size, flags);
}
```

1.6 Development of Requirement Specifications

To check requirements with Klever it is necessary to develop *requirement specifications*. This part of the user documentation describes how to do that. It will help to fix both existing requirement specifications and to develop new ones. At the moment this section touches just rules of correct usage of specific APIs while some things may be the same for other requirements.

In ideal development of any requirements specification should include the following steps:

1. Analysis and description of checked requirements.
2. Development of the requirements specification itself.
3. Testing of the requirements specification.

If you will meet some issues on any step, you should repeat the process partially or completely to eliminate them. Following subsections consider these steps in detail. As an example we consider a requirements specification devoted to correct usage of a module reference counter API in the Linux kernel.

1.6.1 Analysis and Description of Checked Requirements

At this step one should clearly determine requirements to be checked. For instance, for rules of correct usage of specific APIs it is necessary to describe related elements of APIs and situations when APIs are used wrongly. Perhaps, various versions and configurations of target programs can provide APIs differently while considered correctness rules may be the same or almost the same. If you would like to support these versions/configurations, you should also describe corresponding differences of APIs.

There are different sources that can help you to formulate requirements and to study APIs. For instance, for the Linux kernel they are as follows:

- Documentation delivered together with the source code the Linux kernel (directory `Documentation`) as well as the source code of the Linux kernel itself.
- Books, papers and blog posts devoted to development of the Linux kernel and its loadable modules such as device drivers.
- Mailing lists, including Linux Kernel Mailing List.
- The history of development in Git.

Using the latter source you can find out bugs fixed in target programs. These bugs can correspond to common weaknesses of C programs like buffer overflows as well as they can implicitly refer to specific requirements, in particular rules of correct usage of specific APIs.

Technically it is possible to check very different requirements within the same specification, but we do not recommend to do this due to some limitations of software model checkers (*verification tools*). Nevertheless, you can formulate and check requirements related to close API elements together.

Let's consider rules of correct usage of the module reference counter API in the Linux kernel. For brevity we will not consider some elements of this API.

Linux loadable kernel modules can be unloaded just when there is no more processes using them. One should call `try_module_get()` in order to notify the Linux kernel that module is still in use.

bool **`try_module_get`**(struct *module* *module)

Try to increment the module reference count.

Parameters

- **`module`** – The pointer to the target module. Often this the given module.

Returns

True in case when the module reference counter was increased successfully and *False* otherwise.

To give the module back one should call `module_put()`.

void **`module_put`**(struct *module* *module)

Decrement the module reference count.

Parameters

- **`module`** – The pointer to the target module.

There are static inline stubs of these functions when module unloading is disabled via a special configuration of the Linux kernel (**CONFIG_MODULE_UNLOAD** is unset). One can consider them as well, though, strictly speaking, in this case there is no requirements for their usage.

Correctness rules can be formulated as follows:

1. One should not decrement non-incremented module reference counters. Otherwise the kernel can unload modules in use that can result to different issues.
2. Module reference counters should be decremented to their initial values before finishing operation. If this will not be the case one will not be able to unload modules ever.

1.6.2 Development of Requirements Specification

Development of each requirements specification includes the following steps:

1. Developing a model of an API.
2. Binding the model with original API elements.
3. Description of the new requirements specification.

We recommend to develop new requirement specifications on the basis of existing ones to avoid various tricky issues and to speed up the whole process considerably. Also, we recommend you to deploy Klever in the *development* mode (*Local Deployment*). In this case you will get much more debug information that can help you to identify various issues. Moreover, you will not even need to update your Klever installation. Though Web UI supports rich means for creating, editing and other operations with verification job files including specifications, we recommend you to develop requirement specifications directly within *$KLEVER_SRC* by means of some IDE or editor. To further reduce manual efforts using such the workflow, you can temporarily modify necessary preset verification jobs, e.g. to specify requirement specifications and program fragments of interest within `job.json`. Do not forget to not commit these temporary changes to the repository!

Developing Model

First of all you should develop a model of a considered API and specify pre- and postconditions of API usage within that model. Klever suggests to use the C programming language for this purpose while one can use some library functions having a special semantics for software model checkers, e.g. for modeling nondeterministic behavior, for using sets and maps, etc.

The model includes a *model state* that is represented as a set of global variables usually. Besides, it includes *model functions* that change the model state and check for pre- and postconditions according to semantics of the modelled API.

Ideally the model behavior should correspond to behavior of the corresponding implementation. However in practice it is rather difficult to achieve this due to complexity of the implementation and restrictions of verification tools. You can extend the implementation behavior in the model. For example, if a function can return one of several error codes in the form of the corresponding negative integers, the model can return any non-positive number in case of errors. It is not recommended to narrow down the implementation behavior in the model (e.g. always return 0 though the implementation can return values other than 0) as it can result in some paths will not be considered and the overall verification quality will decrease.

In the example below there is the model state represented by global variable **ldv_module_refcounter** initialized by 0. This variable is changed within model functions **ldv_try_module_get()** and **ldv_module_put()** according to the semantics of the corresponding API.

The model makes 2 checks by means of **ldv_assert()**. The first one is within **ldv_module_put()**. It is intended to find out cases when modules decrement the reference counter without incrementing it first. The second check is within **ldv_check_final_state()** invoked by the *environment model* after modules are unloaded. It tracks whether modules decrement reference counters to their initial values before finishing their operation.

```c
/* Definition of struct module. */
#include <linux/module.h>
/* Definition of ldv_assert() that calls __VERIFIER_error() when its argument is not␣
↪true. */
#include <ldv/verifier/common.h>
/* Definition of ldv_undef_int() invoking __VERIFIER_nondet_int(). */
#include <ldv/verifier/nondet.h>

/* NOTE Initialize module reference counter at the beginning */
static int ldv_module_refcounter = 0;

int ldv_try_module_get(struct module *module)
{
    /* NOTE Nondeterministically increment module reference counter */
    if (ldv_undef_int()) {
        /* NOTE Increment module reference counter */
        ldv_module_refcounter++;
        /* NOTE Successfully incremented module reference counter */
        return 1;
    }
    else
        /* NOTE Could not increment module reference counter */
        return 0;
}

void ldv_module_put(struct module *module)
{
```

(continues on next page)

(continued from previous page)

```
    if (ldv_module_refcounter < 1)
        /* ASSERT One should not decrement non-incremented module reference counters */
        ldv_assert();

    /* NOTE Decrement module reference counter */
    ldv_module_refcounter--;
}

void ldv_check_final_state(void)
{
    if (ldv_module_refcounter)
        /* ASSERT Module reference counter should be decremented to its initial value before finishing operation */
        ldv_assert();
}
```

It is worth noting that model functions do not refer their parameter **module**, i.e. they consider all modules the same. This can result to both false alarms and missed bugs. Nevertheless, often it does have sense to do such tricks to avoid too complicated models for verification, e.g. accurate tracking of dynamically created objects of interest using lists. Another important thing is modelling of nondeterminism in **ldv_try_module_get()** by invoking **ldv_undef_int()**. Thanks to it a software model checker will cover paths when **try_module_get()** can successfully increment the module reference counter and when this is not the case.

In the example above you can see comments starting with words **NOTE** and **ASSERT**. These comments are so called *model comments*. They emphasize expressions and statements that make some important actions, e.g. changing the model state. Later these comments will be used during visualization and expert assessment of verification results. You should place model comments just before corresponding expressions and statements. Each model comment has to occupy the only line.

The given API model is placed into a separate C file that will be considered alongside the source code of verified modules. A bit later we will discuss how to name this file and where to place it.

Binding Model with Original API Elements

To activate the API model you should bind model functions to points of use of original API elements. For this purpose we use an aspect-oriented extension for the C programming language. Below there is a content of an aspect file for the considered example. It replaces calls to functions *try_module_get()* and *module_put()* with calls to corresponding model functions **ldv_try_module_get()** and **ldv_module_put()**.

```
before: file ("$this") {
/* Definition of struct module. */
#include <linux/module.h>

extern int ldv_try_module_get(struct module *module);
extern void ldv_module_put(struct module *module);
}

around: call(bool try_module_get(struct module *module))
{
    return ldv_try_module_get(module);
}

```

(continues on next page)

(continued from previous page)

```
around: call(void module_put(struct module *module))
{
    ldv_module_put(module);
}
```

It is not hard to accomplish this aspect file with bindings for static inline stubs of these functions.

The aspect file above contains declarations of model functions. You can place them into a separate header file and include that file into both the C file and the aspect file.

If you will need to keep original function calls, you can use either before/after advices or include those calls directly in advices themselves like in the examples below:

```
/* Original function will be invoked after ldv_func_pre() and its return value will be␣
↪returned eventually. */
before: call(int func(int arg))
{
    ldv_func_pre(arg);
}
```

```
/* Original function will be invoked before ldv_func_post() and its return value will be␣
↪returned eventually.
   Besides, it is available as $res in advice body. */
after: call(int func(int arg))
{
    ldv_func_post(arg, $res);
}
```

```
around: call(int func(int arg))
{
    int ret;
    ldv_func_pre(arg);
    ret = func(arg);
    ldv_func_post(arg, ret);
    return ret;
}
```

Unless you are using options “weave in model aspect” and “weave in all aspects”, you can invoke original functions within model functions. Otherwise, you will have recursion due to those original function calls will be also woven in.

Sometimes it may be quite hard to get function declarations to be used in the aspect file. For instance, it is forbidden to use macros in aspect files while macros may be used in sources. Also, there may be different declarations for the same function depending on configurations. If you will see that your model does not work (e.g. code coverage reports can demonstrate this), you can investigate Weaver’s logs to find valid function declarations. There may be warnings like these:

```
These functions were matched by name but have different signatures:
  source function declaration: void iounmap (void volatile *)
  aspect function declaration: void iounmap (int *)
```

Obviously you need to use at least valid function names. Otherwise, you will not see any warnings. Also, you should take into account that CIF does not issue these warnings for composite pointcuts unless there will be mismatches of original function declarations with their last primitive pointcuts.

You can find more details about development of aspect files and related internals in [N13]. Moreover, you can visit the official project site and read its official documentation.

Description of New Requirements Specification

Bases of requirement specifications are located in JSON files corresponding to projects, e.g. `Linux.json`, within directory *$KLEVER_SRC*`/presets/jobs/specifications`. Also, there is corresponding directory `specifications` in all verification jobs. Each requirements specification can contain one or more C source files with API models. We suggest to place these files according to the hierarchy of files and directories with implementation of the corresponding API elements. For example, you can place the C source file from the example above into *$KLEVER_SRC*`/presets/jobs/specifications/linux/kernel/module.c` as the module reference counter API is implemented in file `kernel/module.c` of the Linux kernel.

Additional files such as aspect files should be placed in the same way as C source files but using appropriate extensions, e.g. *$KLEVER_SRC*`/presets/jobs/specifications/linux/kernel/module.aspect`. You should not specify aspect files within the base since they are found automatically.

As a rule identifiers of requirement specifications are chosen according to relative paths of C source files with main API models. For example, for the considered example it is **kernel:module**. Requirement specification bases represent these identifiers in the tree form.

1.6.3 Testing of Requirements Specification

We recommended to carry out different types of testing to check syntactic and semantic correctness of requirement specifications during their development and maintenance:

1. Developing a set of rather simple test programs, e.g. external Linux loadable kernel modules, using the modelled API incorrectly and correctly. The verification tool should report Unsafes and Safes respectively unless you will develop such the test programs that do not fit your models.
2. Validating whether known violations of checked requirements can be found. Ideally the verification tool should detect violations before their fixes and it should not report them after that. In practice, the verification tool can find other bugs or report false alarms, e.g. due to inaccurate environment models.
3. Checking target programs against requirement specifications. For example, you can check all loadable kernel modules of one or several versions or configurations of the Linux kernel or consider some relevant subset of them, e.g. USB device drivers when developing appropriate requirement specifications. In ideal, a few false alarms should be caused by incorrectness or incompleteness of requirement specifications.

For item 1 you should consider existing test cases and their descriptions in the following places:

- *$KLEVER_SRC*`/klever/cli/descs/linux/testing/requirement specifications/tests/linux/kernel/module`
- *$KLEVER_SRC*`/klever/cli/descs/linux/testing/requirement specifications/desc.json`
- *$KLEVER_SRC*`/presets/jobs/linux/testing/requirement specifications`

For item 2 you should consider existing test cases and their descriptions in the following places:

- *$KLEVER_SRC*`/klever/cli/descs/linux/validation/2014 stable branch bugs/desc.json`
- *$KLEVER_SRC*`/presets/jobs/linux/validation/2014 stable branch bugs`

In addition, you should refer *How to generate build bases for testing Klever* to obtain build bases necessary for testing and validation.

Requirement specifications can be incorrect and/or incomplete. In this case test and validation results will not correspond to expected ones. It is necessary to fix and improve the requirements specification while you will have appropriate resources. Also, you should take into account that non-ideal results can be caused by other factors, for example:

- Incorrectness and/or incompleteness of *environment models*.
- Inaccurate algorithms of the verification tool.
- Generic restrictions of approaches to development of requirement specifications, e.g. when using counters rather than accurate representations of objects.

1.6.4 Using Argument Signatures to Distinguish Objects

As it was specified above, it may be too hard for the verification tool to accurately distinguish different objects like modules and mutexes since this can involve complicated data structures. From the other side treating all objects the same, e.g. by using integer counters when modeling operations on them, can result in a large number of false alarms as well as missed bugs. For instance, if a Linux loadable kernel module acquires two different mutexes sequentially, the verification tool will detect that the same mutex can be acquired twice that will be reported as an error.

To distinguish objects we suggest using so-called *argument signatures* — identifiers of objects which are calculated syntactically on the basis of the expressions passed as corresponding actual parameters. Generally speaking different objects can have identical argument signatures. Thus, it is impossible to distinguish them in this way. Ditto the same object can have different argument signatures, e.g. when using aliases. Nevertheless, our observation shows that in most cases the offered approach allows to distinguish objects rather precisely.

Requirement specifications with argument signatures differ from requirement specifications which were considered earlier. You need to specify different model variables, model functions and preconditions for each calculated argument signature. For the example considered above it is necessary to replace:

```c
/* NOTE Initialize module reference counter at the beginning */
static int ldv_module_refcounter = 1;

int ldv_try_module_get(struct module *module)
{
    /* NOTE Nondeterministically increment module reference counter */
    if (ldv_undef_int() == 1) {
        /* NOTE Increment module reference counter */
        ldv_module_refcounter++;
        /* NOTE Successfully incremented module reference counter */
        return 1;
    }
    else
        /* NOTE Could not increment module reference counter */
        return 0;
}
```

with:

```c
// for arg_sign in arg_signs
/* NOTE Initialize module reference counter{{ arg_sign.text }} at the beginning */
static int ldv_module_refcounter{{ arg_sign.id }} = 1;

int ldv_try_module_get{{ arg_sign.id }}(struct module *module)
{
    /* NOTE Nondeterministically increment module reference counter{{ arg_sign.text }} */
```

(continues on next page)

(continued from previous page)

```
    if (ldv_undef_int() == 1) {
        /* NOTE Increment module reference counter{{ arg_sign.text }} */
        ldv_module_refcounter{{ arg_sign.id }}++;
        /* NOTE Successfully incremented module reference counter{{ arg_sign.text }} */
        return 1;
    }
    else
        /* NOTE Could not increment module reference counter{{ arg_sign.text }} */
        return 0;
}
// endfor
```

In bindings of model functions with original API elements it is necessary to specify for what function arguments it i necessary to calculate argument signatures. For instance, it is necessary to replace:

```
around: call(bool try_module_get(struct module *module))
{
    return ldv_try_module_get(module);
}
```

with:

```
around: call(bool try_module_get(struct module *module))
{
    return ldv_try_module_get_$arg_sign1(module);
}
```

Models and bindings that use argument signatures should be described differently within requirement specification bases. It is recommended to study how to do this on the base of existing examples, say, **kernel:locking:mutex**.

You can find more details about the considered approach in [N13-2].

1.7 Development of Environment Model Specifications

Libraries, user inputs, other programs, etc. constitute an environment that can influence a program execution. It is necessary to provide a model which represents certain assumptions about the environment to verify any program:

- It should contain models of undefined functions which the program calls during execution and which can influence verification results.
- It should correctly initialize external global variables.
- It should contain an **entry-point** function for a verification tool to start its analysis from it. User-space programs have the **main** function that can be used as an entry point, but operating systems and other system software require adding an artificial one.

Our experience shows that bug-finding is possible even without accurate environment models. Still, precise environment models help to improve code coverage and avoid false alarms. It is crucial to provide the accurate environment model considering the specifics of checked requirements and programs under verification to achieve high-quality verification results. It becomes even more essential to provide the appropriate environment model to avoid missing faults and false alarms verifying program fragments.

1.7.1 Environment Model Generator

The environment models generation step follows the program decomposition. Provided program is decomposed into separate independent *program fragments*. Each program fragment consists of several C source files. We call these files below program files.

EMG is a Klever component (plugin) that generates an environment model for a single provided program fragment. It is highly configurable and extendable, so a user needs to prepare its proper configuration to verify a new program.

A JSON file with the requirement specifications base has a section with templates. Such templates contain a *plugins* entry that lists the configuration of different plugins to run. The EMG should always be the first one. See the example in `presets/jobs/specification/Linux.json` in *$KLEVER_SRC*:

```
{
  "templates": {
  "loadable kernel modules and kernel subsystems": {
  "plugins": [
    {
      "name": "EMG",
      "options": {
        "generators options": [
          {"linuxModule": {}},
          {"linuxInsmod": {}},
          {"genericManual": {}}
        ],
        "translation options": {
        "allocate external": false
      }
    }
  ]
}
```

The member with the **options** name contains the EMG configuration. There are descriptions of supported configuration parameters in the following sections of the document.

EMG generates an environment model as the *main C file* and several aspect files intended for weaving their content to program files. We refer to these output files as aspect files. Each aspect file contains the code to add at the beginning of a program file or its end and a description of function calls and macros to replace with models.

EMG generates environment models using the provided source code given as a project build base and *specifications*. Specifications are C files or JSON files with models in C or DSL languages. We distinguish specifications and environment models:

- The environment model is a file in an intermediate EMG notation or in C. The former is a file in the internal representation which is called an *intermediate environment model (IEM)*. The latter is called the *final environment model (FEM)*.
- Environment model specifications can describe IEMs for specific program fragments, models' templates, parts, or even configuration parameters. Specifications are always prepared or modified by hand and provided as input to EMG.

The Klever presets directory has the *specifications* directory. It contains all specifications for different programs and components. EMG does not require pointing to specific files at providing specifications. It searches for all specifications in the directory and applies only relevant ones. Files of specifications for the EMG plugin have a particular naming policy. Their names always end with a suffix that distinguishes their utilization. These suffixes are described below.

EMG Components

EMG has a modular architecture, so one needs to know it to configure the plugin and/or even extend it properly. The picture below shows its components:

The input of the EMG component includes the configuration parameters (plugin configuration), specifications and the build base with the source code and its meta-information.

The output of the component consists of several environment models for the given program fragment.

There are three main components in the EMG that a user must appropriately configure: Generator pipeline, Decomposer, and Translator. These components are considered below in detail, but we give information about their primary functions in this section.

The Generator pipeline runs several generators one by one. Generators yield parts of the IEM. Generated parts are independent and form the IEM as a parallel composition.

Decomposer separates the IEM into several simplified parts that can be verified independently. This step is optional.

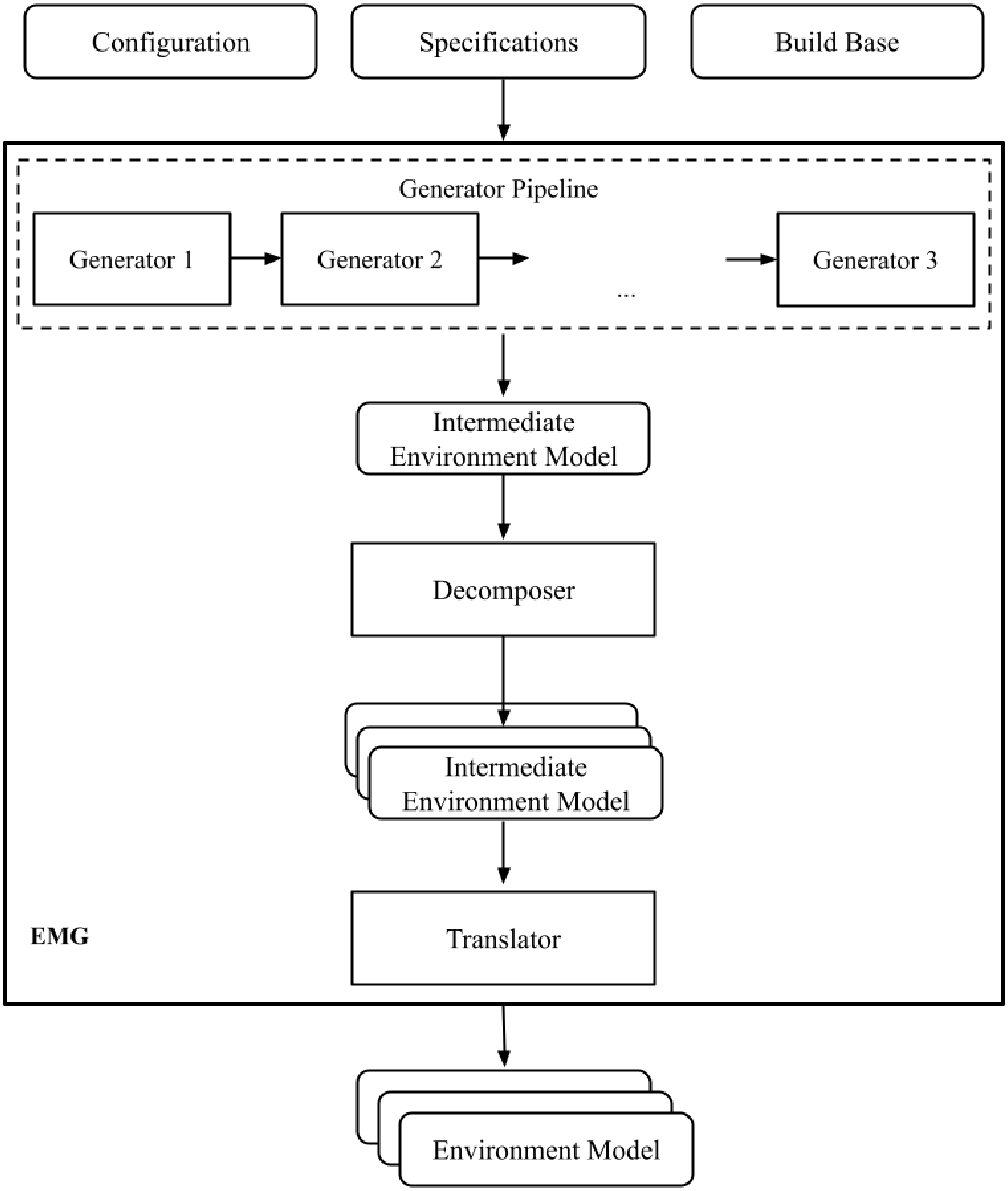
Translator prepares the C code based on the provided IEM. It applies many simplifications to the input model. If there are several input models, several Translator instances are executed and generated FEMs are independent.

EMG Configuration

There are the following main configuration parameters of the EMG plugin:

Table 1.1: Main EMG configuration parameters.

Configuration Parameter	Value Type	Default Value	Description
specifications set	String	None	The value is an identifier of the specification set. For example, an identifier can correspond to a particular Linux kernel version. The LinuxModule generator expects one of the following values: 2.6.33, 3.2, 3.14, 3.14-dentry-v2, 4.6.7, 4.15, 4.17, 5.5. The parameter can be provided directly in the `job.json` file.
generators options	Object	None	The list defines the sequence of generators in the Generators pipeline. For example: “generators options”: [{“linuxModule”: {}}, {“linuxInsmod”: {}}, {“genericManual”: {}}]
translation options	Object	None	An object with configuration parameters for Translator.
single environment model per fragment	Bool	true	The false value activates Decomposer. It is described in a separate section as its extra configuration parameters. This parameter is required to be set in `job.json` directly.
dump types	Bool	false	The property is intended for debugging. Generate a file *type collection.json* with the list of imported types.
dump source code analysis	Bool	false	The property is intended for debugging. Generate files `vars.json`, `functions.json`, `macros.json`.


Configuration
Specifications
Build Base
Generator Pipeline
Generator 1
Generator 2
...
Generator 3
Intermediate Environment Model
Decomposer
Intermediate Environment Model
EMG
Translator
Environment Model

1.7.2 Intermediate Environment Model

EMG generates an IEM before translating it to the C language. The model is combined as a parallel composition from parts prepared by generators. The model also can be fully designed by hand and provided directly to the EMG using a specific generator (genericManual). We refer to such input files as *user-defined environment model specifications (UDEMS)*. Specifications for other generators include only templates or additional information to generate parts of IEMs.

IEMs and UDEMSes have the same notation. It is a JSON file. However, the structure of files containing these two kinds of models is slightly different. We consider the notation of only UDEMSes below because such specifications include IEMs.

Structure of UDEMS

A root is an object that maps *specification set identifiers* (related to configuration property *specifications set* mentioned above) to specifications itself. Specification sets are intended to separate models for different versions of a program. The specification contains IEMs meant for particular program fragments. The example below shows the organization of a file with a UDEMS:

```
{
  "5.5": [
    {
      "fragments": [
        "ext-modules/manual_model/unsafe.ko",
        "ext-modules/manual_model/safe.ko"
      ],
      "model": {}
    }
  ]
}
```

Program fragment identifiers are generated automatically by Klever at verification. One can get these names from attributes of plugin reports or verification results in the web interface. Also, the PFG component report contains the list of all generated program fragments.

The *model* value is an IEM provided to the EMG.

We do not give the precise theoretical semantics of the notation in the document. You can find them in the following papers [Z18], [N18], [ZN18]. Instead, we describe the semantics intuitively by making analogies with program execution. We say about execution and running of processes, but even in the C code, IEM cannot be ever executed. It is intended only for analysis by software verification tools, so we say this just to avoid overcomplications.

Each IEM is a parallel composition of transition systems called *processes*. Each transition system can be considered as a thread executed by an operating system. The model contains *environment processes*. Each transition system has a state and can do actions to change the state. The state is defined by values of labels. Intuitively labels can be considered as local variables on the stack of a process.

A model consists of a main process, environment processes and function models. Both three are described with process descriptions, but semantically they are different. The main process is like a thread that acts from the very beginning of a combination of a program and environment model. It may trigger execution of a program or send signals to activate environment processes. While a program code is executed, it may call functions that are replaced by models. Function models are not processes or threads in any sense, they just act within the same scope, they can send signals to environment processes but cannot receive any.

Environment processes exist from the very beginning of execution as the main process does. But any such process expects a signal to be sent to it for activation before doing any other activity. Signals are described below in more detail.

Each label has a C type. Any process can do block actions and send/receive signals. A block action is a C base block with C statements over program fragment global variables and labels. Signals pass values of labels and synchronize the sequence of actions between processes.

Process Actions

A process performs actions. There are actions of following kinds:

- block actions describe operations performed by the model.
- send/receive actions establish synchronization.
- jump actions help to implement loops and recursion.

The behavior of an environment model is often nondeterministic. Let's consider a typical combination of an environment model with a program fragment source code. The semantics will be the following:

- The main process starts doing its actions from the very beginning first.
- It would either call a function from the program fragment or send an activating signal to any of environment model processes.
- The process transfer follows the rendezvous protocol:
  - The sender waits until there is a receiver in the state when it can take a receiving action.
  - Then the receive happens in no time. Nothing can happen during the receive.
  - If a receiver or a sender may do any other action instead of signal sending, they are allowed to attempt it leaving the other process still waiting. But if a process has the only option (sending or receiving a signal), then it cannot bypass it.
  - If there are several possible receivers or dispatchers, then the two are chosen randomly.
- If there is a signal receiver that belongs to environment processes, it begin doing his actions. So, there are the main process and recently activated environment processes doing their actions in parallel with each other.
- If a process attempts doing its base block action, then it waits until it is executed before doing next actions. The code may contain calls of functions defined in a program fragment. Such code can call undefined functions for which there are function models in turn. When execution reach the function call with an existing function model, the following switch of execution happens:
  - The host process which is doing his base block action still cannot attempt any other actions.
  - The execution of the source code of the base block is paused.
  - A new function model begins its execution in the same context.
  - The function model attempts doing its actions as any other process. It may do base block execution, send signals, etc.
  - The last action of the function model should contain the return statement with values provided back to the paused code as any function does after its finishing.
  - The execution of the source code of the base block is resumed.
  - Other processes do their stuff in parallel during the described procedure as usual.

We propose a simple DSL to describe possible sequences of actions that can be performed by the environment.

The order of actions is specified in the *process* attribute entry of a process description (considered below) using a simple language:

- <name> is a base block action;
- (!name) or (name) is a signal receiving. Where (!name) means that the process waits for a signal to start doing actions. The (name) is a signal receiving action that can be used in any place except as the first action.
- [name] is a signal sending action.
- {jump} is an auxiliary jump action that just specifies a new sequence of actions to do. Each jump action has its process entry. Jumps do not form a stack: a process does not return to an interrupted action sequence.

Order of actions is described with the help of two operators:

1. "**.**" is a sequential combination operator. Actions *a . b* combined this way mean *b* follows *a*.
2. "**|**" is a non-deterministic choice operator. Only one action of combined ones will be selected for *a* | *b*. But verification tools analyse both options as possible alternatives.

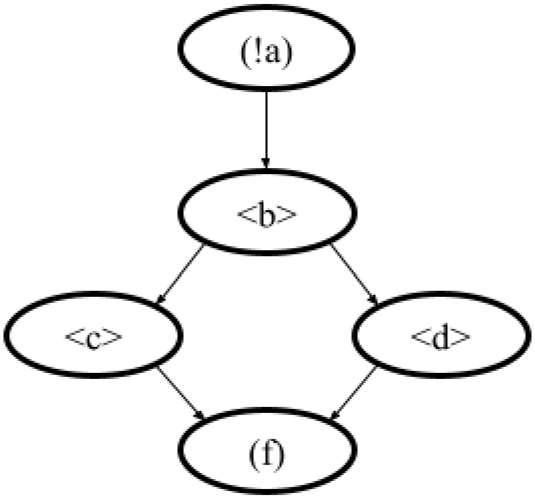
A sequential combination operator has a higher priority than choices. Parentheses in expressions can also be used, but do not confuse them with signal receiving.

All actions can have conditions or guards (look at the table in the next section). But they work differently in different situations:

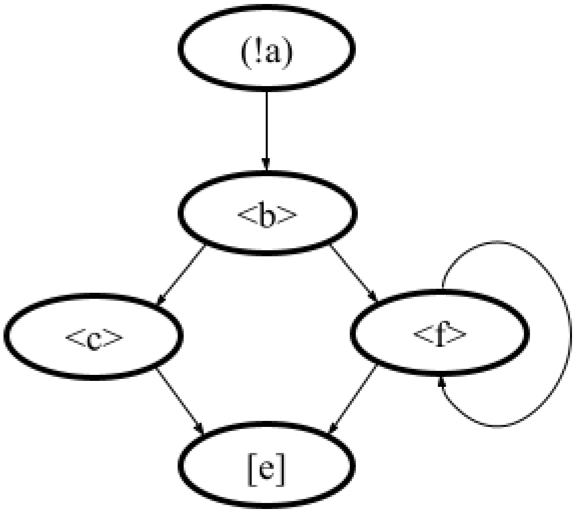
- **Receive:** condition is a guard to check whether the signal can be received.
- **Dispatch:** skip the action if the guard is not feasible.
- **Jump:** conditions are not supposed to be added.
- **Base block:**
  - **In choice operator:** do not choose the whole branch of actions. Let's consider an example *(<a>.<b>.<c> | <d>)*. Imagine, *a*, *b* and *c* have conditions. Then if the *a*'s condition is false, both *a*, *b*, *c* cannot be chosen. The same if the *d* 's condition is infeasible (*d* will be skipped). But if the *b*'s condition is not evaluated to be true, *a* can be chosen, *b* will be just skipped, and *d* will be done.
  - **In sequential combination:** skip the action if the guard is evaluated to false.

There are several examples of actions order written down using the proposed notations and corresponding state machines describing that order:

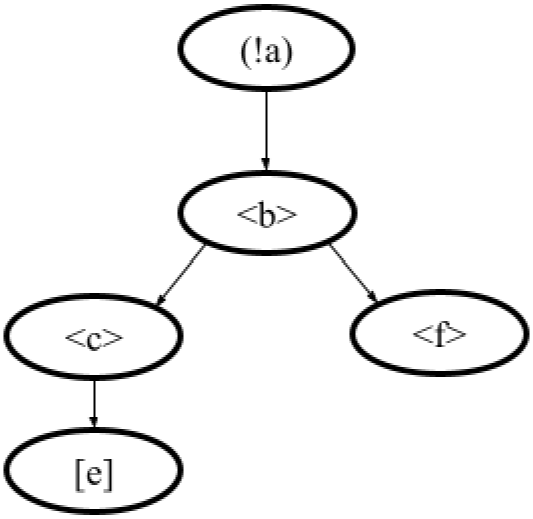
The first example: process = (!a).<b>.(<c> | <d>).(f)



The second example: process = (!a).<b>.(<c> | {d}).[e] where jumping action is d = <f>.{d} | [e].



The third example: process = (!a).<b>.(<c>.[e] | <f>)



The example demonstrates the usage of conditions in first base block actions of the choice operator.

A user may need to attempt an action several times in a row. There is an additional form to describe such repeated actions. Imagine, there is a base block action *x*, then the process should contain it in the following form to repeat it twice: *<x[2]>*. It is possible to provide only positive integer numbers in square brackets (if by some reason you would like to use 0, you should likely get rid of this action at all).

Jumps have the same notation but the semantics is a bit different. Here an integer number means the number of jumps that will be done on a single path through this action. For instance, *{my_jump[2]}* will return to this jump twice. There is an another kind of guards to restrict passing through a jump: *{my_jump[%flag%]}*. This jump depends on the evaluation of the *%flag%* label that serves as a guard. A user may choose any defined process label as a guard. It is worth noting that one has to change the *%flag%* label manually in other actions of the process, so that it should become 0 at the end of the day. The current implementation stops execution when the guard label is evaluated to 0. Thus, following actions will not be executed at all. Taking all this into account, it may be better to use dedicated counters and conditions for restricting the number of possible iterations through jump actions.

Model Description

Let us consider the notation of the UDEMSes.

Each process has an identifier consisting of a category and name. Categories of environment processes can be shared. A category reflects which part of an environment is modeled by specific processes. Processes' identifiers should be unique, but names can be reused.

Note that names of models, processes and some actions are used by the web interface and it is very important to keep them short and clear.

The root object has the following attributes:

Table 1.2: IEM root members.

Name	Value type/default value	Description	Required
name	string/"base"	The name of the model.	No
main process	*Process description* object/*null*.	The main process describes the first process of the environment model that does not wait for any registering signals.	No
environment processes	Value is an object that maps process identifiers (a category and process name separated by "/" symbol) to process descriptions. Process identifiers are used in attributes. A category and process name should be C identifiers. Example: { "category1/name1": {process description}, "category2/name2": {process description} }	Names are identifiers of processes described in values.	No
functions models	Value is an object that maps enumerations of function names to corresponding process descriptions: {"name1, …, nameN": {process description}, "name": {process description}}	A name of the attribute is a string which is an enumeration of function names. All these functions will be replaced by the same model generated from the provided process description.	No

The model's name is not necessary but the EMG component can generate several models per program fragment and such models would have distinguished names.

An example of a UDEMS structure is given below. Processes' descriptions are omitted.

```
{
  "name": "Example",
  "main process": {},
  "environment processes": {
    "platform/main": {},
    "platform/PowerManagement": {}
  },
  "functions models": {
    "f1, f2": {}
  }
}
```

A process description has the following attributes:

Table 1.3: Process description members.

Name	Value type/default value	Description	Required
comment	Arbitrary string	The comment is used at error-trace visualization. It should describe what the process implements.	Yes
process	Process transition relation (see its description below).	Transition relation describes the possible order of actions performed by the process.	Yes
actions	The object maps action names to action descriptions. Action names should be C identifiers.	Actions describe the behavior of the environment model.	Yes
labels	The object maps label names to label descriptions. Label names should be C identifiers. {“var”: {…}, “ret”: {…}}	Labels represent the state of a process.	No
headers	A list of relative paths to header files: [“stdio.h”, “pthread.h”]	Headers are included in the main C file of an environment model to bring type definitions and function declarations to the main C file of the FEM.	No
declarations	The option maps names of program source files or *environment model* (meaning the main C file) to maps from C identifiers to declarations to add. C identifiers are used to combine declarations from different process descriptions at translation. If identifiers from different process descriptions match, then only one value is selected for the main C file. {“dir/name.c”: {“func”: “extern void func(void);”}, “environment model”: {“func”: “void func(void);”}}	Declarations are added to the beginning of the given files (program files or the main C file).	No
definitions	The object maps names of program fragment files or *environment model* (mean the main C file) to maps from C identifiers to definitions of functions to add or paths to C files to inline. In the case of a C file, whole its content will be weaved into the program file or main C file. To generate the wrapper for a static function in the program fragments’s source code, one can use a shorter form with members declaration and wrapper members. The former is the declaration of the target static function, the latter is the name of the wrapper to generate. { “file.c”: { “myfunctions”: “linux/file.c”, “wrapper”: [“void wrapper(void) {”, “func();”, “}”], “callback”: {“declaration”: “static void callback(void)”, “wrapper”: “emg_callback”} } }	Definitions work the same way as declarations, but definitions are multi-line and added after declarations to files of a program fragment or the main C file.	No
peers	The map from process identifiers to lists of action names. “peers”: {“c/name”: [“register”]}	The member describes which processes are connected with this one. Keys of the map list names of processes that can send signals to the process or receive signals from it. Values enumerate corresponding sending and receiving actions.	No

There is an example of a process description with simplified values below:

```
{
  "comment": "Invoke platform driver callbacks.",
  "process": "(!register).<probe>.(<ok>.{pm_jump} | <fail>).<free>.(deregister)",
  "actions": {
      "deregister": {},
      "fail": {},
      "free": {},
      "ok": {},
      "pm": {},
      "pm_jump": {},
      "probe": {},
      "register": {},
      "release": {}
  },
  "labels": {},
  "headers": ["linux/platform_device.h"],
  "declarations": {
      "environment model": {
          "get_dev_id": "const struct platform_device_id *get_dev_id(struct platform_
↪driver *drv);"
      }
  },
  "definitions": {
      "environment model": {
          "get_dev_id": [
              "const struct platform_device_id *get_dev_id(struct platform_driver *drv) {
↪",
              "\treturn & drv->id_table[0];",
              "}"
          ]
      }
  }
}
```

We will describe labels and actions below with more discussion. The *headers* member has a single header to add. It is necessary to allocate memory and dereference pointers to `platform_driver` and `platform_device` structures. *Declarations* and *definitions* members introduce a function `get_dev_id()` that can be used in actions. Its definition and declaration will be added to the main C file of the FEM. We suggest users to implement more complicated functions in separate C files and provide a path to them instead of the list of strings.

Labels

Each label has a type and value just as variables. A label can have any C type respecting the scope of the main C file. An initial value for the label should be provided directly as a string. It can refer to any variables from the scope of the main C file.

An object that describes a label has the following attributes:

Table 1.4: Label description members.

Name	Value type/default value	Description	Required
declaration	Declaration of the C type, e.g.: void *ptr	The attribute stores the type of the label.	Yes
value	String	String with an optional initial value of the label respecting its type.	No

There is an example of labels descriptions for the example provided above.

```
"labels": {
  "driver": {"declaration": "struct platform_driver \*s"},
  "device": {"declaration": "struct platform_device \*device"},
  "msg": {"declaration": "pm_message_t msg"},
  "ret": {"declaration": "int a", "value": "ldv_undef_int_nonpositive()"}
}
```

Jump Actions

Before we will consider how these labels are used in actions, let us consider the order of actions and provide a description of the *pm_jump* jump action.

The sequence of actions provided within a process attribute can be reduced to another sequence implemented in jump action. Its description can have the following attributes.

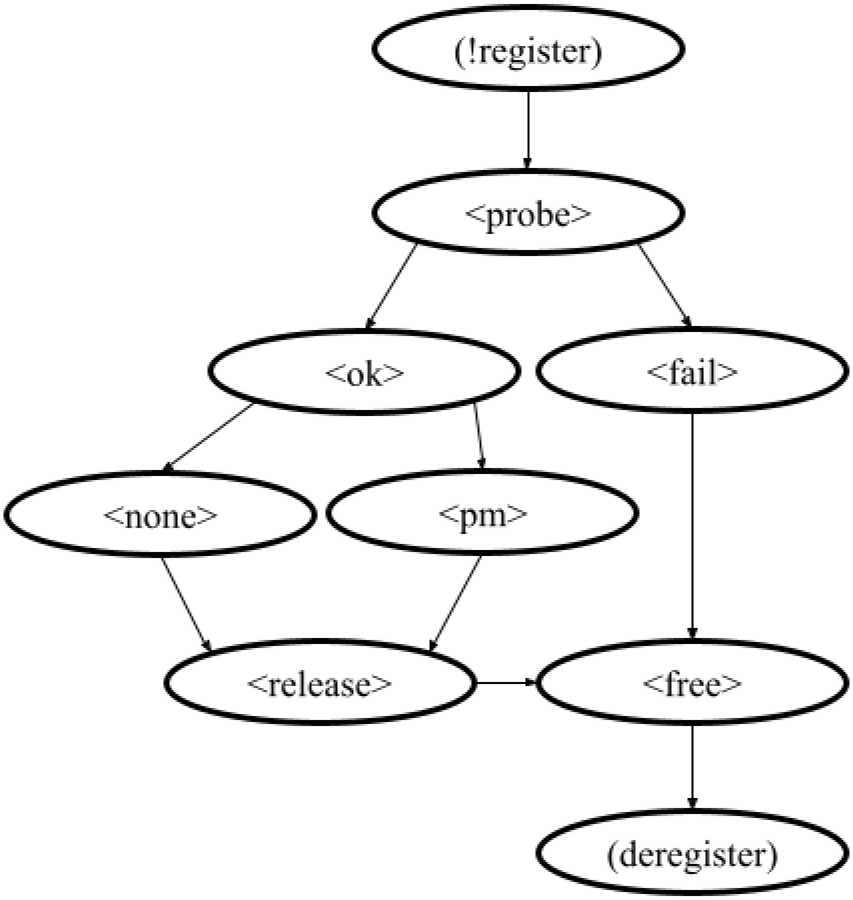
Table 1.5: Jump action description.

Name	Value type/default value	Description	Required
comment	String with the action description.	Comments help users to understand error traces better.	Yes
process	Process transition relation (see its description below)	Transition relation of the subprocess.	Yes

The code below demonstrates the action description of the *pm_jump* action for the example provided above.

```
"pm_jump": {
  "comment": "Run PM callbacks or just remove the driver.",
  "process": "(<pm> | <none>).<release>.<free>.(deregister)"
}
```

Together with the process member of the process description they set the following order of actions:


(!register)
<probe>
<ok>
<fail>
<none>
<pm>
<release>
<free>
(deregister)

Signaling Actions

Signal dispatches and receives have parameters and names. A signal can be passed if there are two processes: one should have a dispatch action, and another should have a receiving one. Names of actions, the number of parameters, and their types should be the same.

Currently, the implementation of EMG does not support arbitrary signal exchange between processes as such models would be too complicated for verification tools. An environment process can receive signals only as a first action and as the last action. The first received signal is called registration and the last one is deregistration. A function model cannot receive signals but can send them anytime.

Signaling action description can have the following attributes:

Table 1.6: Signal action description.

Name	Value type/default value	Description	Required
comment	String with the action description.	Comments help users to understand error traces better.	Yes
condition	The same as in conditions of base block actions. It is also allowed to use incoming parameters of the signal at receive actions: use *$ARG1*, . . . , *$ARGN* expressions.	The condition restricts the acceptance of signals with the proper name but unexpected values.	No
parameters	A list of label names to save received values or send values from. [“%ret%, “%struct%”]	Labels to save or send data.	Yes
savepoints	It is a map from savepoint names to their descriptions. {“name”: {. . . }}	Savepoints are used at decomposition, and they are considered in the following sections in detail. Any action can have this attribute, but it must be the first one in the process. All savepoints across all environment model processes should have unique names. You should use short names for savepoints as they are shown in the web-interface. Savepoint descriptions are considered below.	Not
require	The object has *processes* and *actions* attributes. The latter lists actions required for this one. Processes contain a map from process identifiers to True/False values that means inclusion or exclusion of these processes. It is not necessary to include the identifier of the process which is a host to the action with the *require* attribute. Actions contain a map from process identifiers to corresponding lists of names of actions that are required. {“require”: {“actions”: {“c/p1”: [“probe”, “success”]}, “processes”: {“c/p1”: true}}}	The attribute says that the action requires another process that should have specific actions in turn. This field is used only at the decomposition of models, which is considered in the following chapters.	No

The examples of register and deregister action descriptions from the example above are given below:

```
"register": {
  "comment": "Register the platform callbacks in the kernel.",
  "parameters": ["%driver%"]
```

(continues on next page)

(continued from previous page)

```
},
"deregister": {
  "comment": "Finish platform callbacks calling.",
  "condition": ["%driver% == $ARG1"],
  "parameters": ["%driver%"]
}
```

The registering action does not have any condition and just saves the received pointer to the platform_driver structure to the driver label. The deregistering action has a guard that checks that the deregistration is performed for the already registered device.

Base Block Actions

Base blocks contain statements in the C programming language. These statements over labels are used to compose the code of a FEM. A user may call any functions, use any global variables and labels of the process but concerning the scope of the main C file. The EMG does not resolve scope issues for you, and to add more variables, types, or functions to the file, one should include or implement additional headers and maybe wrappers of static functions.

Base block action descriptions have the following attributes:

Table 1.7: Block action description.

Name	Value type/default value	Description	Required
comment	String with the action description.	Comments help users to understand error traces better. It is used for error-trace visualization.	Yes
condition	String with a boolean statement over global variables or labels. % symbols enclose label names. "%ret% == 0 && %arg% != 0"	If the condition is feasible, then a verifier can go analyzing the action. If it is infeasible and not the first action of a sequence which is an option of the choice operator, then the action is skipped, and the following is attempted. If the action is the first in a sequence considered as an option of a choice operator, then the whole series is deemed to be unfeasible. Example 1: <a>.<b>.<c> If <b> has an invalid condition, then <b> is just skipped. Example 2: <a>.<b> | <c>.<d> If the <a> action's condition is false then <a>.<b> branch cannot be chosen at all.	No
statements	List of strings with statements to execute over global variables or labels. % symbols enclose label names. ["%one% = 1;", "%ret% = callback(%one%);", "ldv_assume(%ret%);"]	Statements describe state changing. There are just strings with the C code that can call functions from the program fragment or auxiliary C files, access or modify labels and global variables.	No
trace relevant	Bool	True if the action should always be shown in the error trace. If it is false, then in some cases error traces would hide the action.	Yes

Each base block is independent. Its source code should be correct. Avoid leaving open brackets, parentheses, or incomplete operators. It is also forbidden to declare new variables in base blocks.

To use the variables and functions from the program, one needs to include header files as other files of the program fragment do. There are several ways to do it:

1. Add required headers to the *additional header* configuration parameter. These headers will be added to all output models. For this purpose, you may create a separate header file in the specifications directory and include this single file.
2. Add headers to the “headers* attribute of a specific process in the UDEMS. This approach works only using genericManual and linuxModule generators.

The default value of *additional header* configuration parameter lists several files. Find them in the last section devoted to Translator. Inspect them before writing specifications. There are helpful functions there to:

- allocate and free memory;
- insert assumptions in the code;
- initialize undefined values of certain types to implement non-deterministic behavior;
- create and join threads in parallel models.

Sometimes entry points that should be called by the environment models are implemented as static functions. Implement wrappers using *definitions* and *declarations* members of a process description in the case.

There are several auxiliary expressions allowed in base block statements:

- **$ALLOC(%*labelname*%);**
  Allocate memory according to the label type size (the label is expected to be a pointer) using `ldv_xmalloc()` function.
- **$UALLOC(%*labelname*%);**
  Allocate memory according to the label type size (the label is expected to be a pointer) using `ldv_xmalloc_unknown_size()` function.
- **$ZALLOC(%*labelname*%);**
  Allocate memory according to the label type size (the label is expected to be a pointer) using `ldv_xzalloc()` function.
- **$FREE(%*labelname*%);**
  Free memory by `ldv_free()` function.

It is allowed to use function parameters when describing statements and conditions of function models. To do that one may use expressions *$ARG1*, *$ARG2*, etc.

Environment models are connected with requirement specifications. There are two main functions to initialize the model state of requirement specifications and do final checks:

- `ldv_initialize();`
- `ldv_check_final_state()`.

Read about them in the tutorial related to the requirement specifications development. Remember that you may implement more functions that connect requirements with environment models. Just implement proper header files to use them in your models.

Another issue is source code weaving. Requirement specifications and function models in IEMs replace function calls and macro expansion by corresponding models. But functions in IEM and requirement specifications are never replaced this way. Keep it in mind developing your specifications.

There are descriptions of the block actions for the example considered above:

```
"probe": {
  "comment": "Probe the device.",
```

(continues on next page)

(continued from previous page)

```
  "statements": [
      "$ALLOC(%device%);",
      "%device%->id_entry = get_dev_id(%driver%);",
      "%ret% == %driver%->probe(%device%);"
  ]
},
"ok": {
  "comment": "Probing successful, do releasing.",
  "condition": ["%ret% == 0"]
},
"fail": {
  "comment": "Probing failed.",
  "condition": ["%ret% != 0"]
},
"free": {
  "comment": "Free allocated memory.",
  "statements": ["$FREE(%device%);"]
},
"pm": {
  "comment": "Do suspending, then resuming.",
  "statements": [
      "%ret% = %driver%->suspend(%device%, %msg%);",
      "ldv_assume(%ret% == 0);",
      "%ret% = %driver%->resume(%device%);",
      "ldv_assume(%ret% == 0);"
  ]
},
"none": {
  "comment": "Skip PM callbacks."
},
"release": {
  "comment": "Probing successful, do releasing.",
  "condition": ["%ret% == 0"],
  "statements": ["%driver%->release(%device%);"]
}
```

Statements in the actions just contain the C code where labels are used instead of variables and *$ALLOC*/*$FREE* expressions replace the memory allocation and releasing. There are *$UALLOC* to allocate a region of memory with an undefined size and *$ZALLOC* to allocate zeroed memory with a size calculated by *sizeof*. There are calls of `get_dev_id()` and `ldv_assume()` functions there. The first one is defined in declarations and definitions entries. The second one is defined in the `common.h` header which is likely to be included to any UDEMS.

Pay attention to condition names. Actions that are used in the choice operators may have conditions to avoid releasing after unsuccessful probing. But the none action does not have both conditions and statements. It is an auxiliary action that allows it to go to release after an unsuccessful probing skipping the suspend/resume callbacks.

1.7.3 Environment Generator Pipeline

The environment Generator pipeline currently allows using four generators:

- **linuxInsmod** – the generator for calling `init()/exit()` functions of Linux modules.
- **linuxModule** – the generator for calling callbacks of Linux kernel modules and subsystems.
- **genericFunctions** – the generator that allows analyzing independently separate entry point functions provided by a user.
- **genericManual** – the generator that allows a user to completely set the environment model by providing a UDEMS specification.

A user must choose at least one of them by setting the *generators options* configuration parameter.

LinuxInsmod Generator

The generator supports the generation of the main process for an IEM. The main process includes calls of the following functions found in the provided program fragment:

- module initialization functions,
- subsystem initialization functions,
- module exit functions.

Provided program fragment can contain several Linux kernel modules or/and subsystems. The generator prepares a model with an appropriate order of calling all found functions listed above, respecting successful and failed initializations.

Table 1.8: Configuration parameters of linuxInsmod generator.

Configuration Parameter	Value Type	Defaul Value	Description
kernel initialization	List	[“early_initcall”, “pure_initcall”, “core_initcall”, “core_initcall_sync”, “postcore_initcall”, “postcore_initcall_sync”, “arch_initcall”, “arch_initcall_sync”, “subsys_initcall”, “subsys_initcall_sync”, “fs_initcall”, “fs_initcall_sync”, “rootfs_initcall”, “device_initcall”, “device_initcall_sync”, “late_initcall”, “late_initcall_sync”, “console_initcall”, “security_initcall”]	A list of macros is used to provide subsystem initialization functions to the Linux kernel. The generator searches for them to find entry points.
init	str	module_init	The macro is used to provide the module initialization function to the Linux kernel. The generator searches for macros to find entry points.
exit	str	module_exit	The macro used to provide module exit function to the Linux kernel. The generator searches for macros to find entry points.
kernel	bool	False	The generator assumes that the provided program fragment can contain subsystem initialization functions if the flag is set.

LinuxModule Generator

The generator generates environment processes and function models for program fragments containing Linux kernel modules and subsystems. The generator requires two kinds of specifications for its work:

- Interface callback specifications (file names end with *interface spec* suffix) – specifications describe the interface of certain callbacks.
- Event callback specifications (file names end with *event spec* suffix) – specifications of this type have the format very close to the structure of IEMs but extend it a bit. Event specifications describe the part of the environment model that calls callbacks of a particular type.

TODO: Describe formats

TODO: Describe algorithms

Table 1.9: Configuration parameters of linuxModule generator.

Configuration Parameter	Value Type	Defaul Value	Description
action comments	Obj	{ “dispatch”: { “register”: “Register {} callbacks.”, “instance_register”: “Register {} callbacks.”, “deregister”: “Deregister {} callbacks.”, “instance_deregister”: “Deregister {} callbacks.”, “irq_register”: “Register {} interrupt handler.”, “irq_deregister”: “Deregister {} interrupt handler.” }, “receive”: { “register”: “Begin {} callbacks invocations scenario”, “instance_register”: “Begin	This object contains default comments for particular actions that do not have them.

GenericFunctions Generator

The generator helps to start using Klever with a new program. A user provides a list of function names to call with undefined parameters. Such a generator helps get a relatively simple environment model to configure and go through all preparation Klever for verification.

Table 1.10: Configuration parameters of genericFunction generator.

Configuration Parameter	Value Type	Defaul Value	Description
functions to call	List	[]	It is a list with strings containing names of functions or Python regular expressions to search these names.
prefer not called	Bool	False	If there are functions with the same name, the model would call that one that is not called in the program fragment.
call static	Bool	False	Allows calling static functions. By default, provided static functions are ignored.
process per call	Bool	False	Generate a separate process per a function call. It might be very helpful at searching data races.
infinite calls sequence	Bool	False	Call functions in an endless loop.
initialize strings as null-terminated	Bool	False	Create arbitrary null-terminated strings if a function expects such a parameter.
allocate external	Bool	True	Use a specific function to mark variables for the CPAchecker SMG verifier as external memory.
allocate with sizeof	Bool	False	Allocate the memory by calculation of sizeof value for structures. If it is disabled, then the generator uses a specific function returning an undefined pointer.

GenericManual Generator

It is the most precise generator that does not generate anything. It expects a UDEMS specification to produce an environment model. It can be used alongside the previously mentioned generators to combine automatically-generated models with parts developed manually or even replace certain automatically generated parts with manually adjusted versions.

Specifications for the generator have names with *user model* suffixes.

Table 1.11: Configuration parameters of genericManual generator.

Configuration Parameter	Value Type	Default Value	Description
enforce replacement	Bool	True	If the provided IEM and UDEMS have the same process and the flag is true, then the process description from the UDEMS will be used.
keep entry functions	Bool	False	Suppose the main process of an IEM is replaced by one of a UDEMS. In that case, the declarations and definitions will be added to the model from the deleted description. It is helpful not to write wrappers of static functions manually.

1.7.4 Environment Model Decomposition

The EMG has a component for decomposing large and complicated IEMs into simpler ones. The insufficient scalability of verification tools is a reason to perform such decomposition.

The Decomposer component implements two kinds of tactics:

- **Process decomposition** – it is how each process of an IEM can be divided into several so-called *scenarios*.
- **Scenario selection** it is the way how scenarios are combined to get simplified environment models. Original processes can be replaced by scenarios or left as is at this stage.

A scenario is a simplified process that can take fewer actions than the original process. Each process can be split into scenarios if there are choice operators, savepoints (discussed below), or jumps.

Savepoints

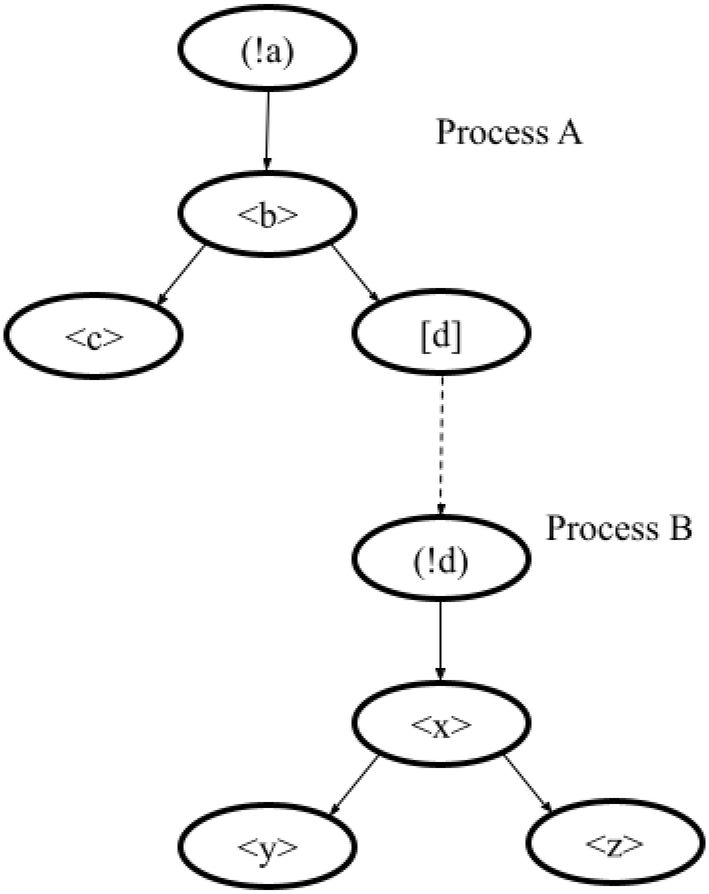
Imagine, that there is a same model illustrated in the picture below. There are two processes A and B. The process A activates the B process.

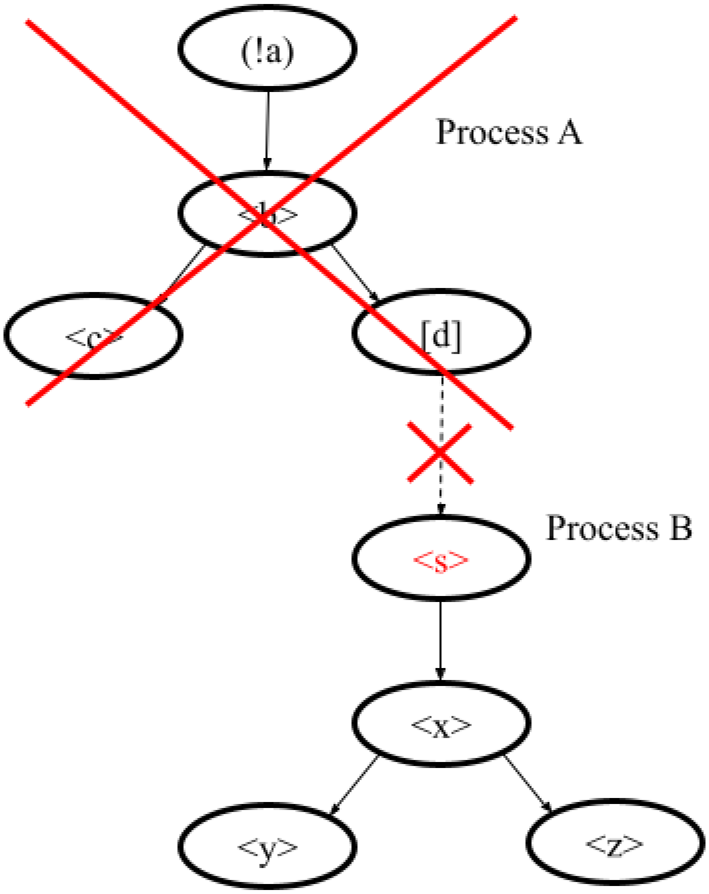
Then imagine that the action *d* has a savepoint *s*. Then after decomposition the B process becomes a new main one and the A process is deleted in this case.

Description of each savepoint for a given action should follow the following table:

Table 1.12: Savepoint description.

Name	Value type/default value	Description	Required
comment	String	Comments help users to understand error traces better.	Yes
statements	A list of strings	Statements contain the code of the process initialization if the process with the savepoint becomes the main one. Values should be provided as for the same attribute of block actions.	Not
require	It is the same map which is already described for the attribute with the same name used in actions.	Requirements allow to restrict actions and processes that should be included to models with the savepoint.	Not


(!a)
Process A
<b>
<c>
[d]
Process B
(!d)
<x>
<y>
<z>


(!a)
Process A
<b>
<c>
[d]
Process B
<s>
<x>
<y>
<z>

There is an example of a savepoint attached to the registering action considered in the section related to IEM and UDEMS:

```
"register": {
  "comment": "Receive a container.",
  "parameters": ["%driver%"],
  "savepoints": {
      "s1": {
          "comment": "Allocate memory for the driver with sizeof.",
          "statements": ["$ALLOC(%driver%);"]
      },
      "s2": {
          "comment": "Allocate memory for the driver without sizeof.",
          "statements": ["$UALLOC(%driver%);"]
      }
  }
}
```

Names *s1* and *s2* are used for savepoints in the example, so other savepoints should borrow other names. These savepoints can replace the main process of the IEM and allocate memory for the driver structure instead of receiving it from outside (its initialization is omitted for simplicity, it is possible to extract it into a separate C function and invoke it here to make savepoints code shorter).

A scenario may have a savepoint. It means that the scenario can be used as a replacement of the main environment process only. In this case, the origin process from which the scenario is generated is removed from the model to which the scenario is added as the previous main one also.

Decomposition Tactics

There are three implementations of process decomposition tactics. The default one is used if the value of the *scenario separation* configuration property is None (find the description in the table below). The default tactic does not modify actions. But instead, it creates a scenario with the origin actions and different scenarios with savepoints.

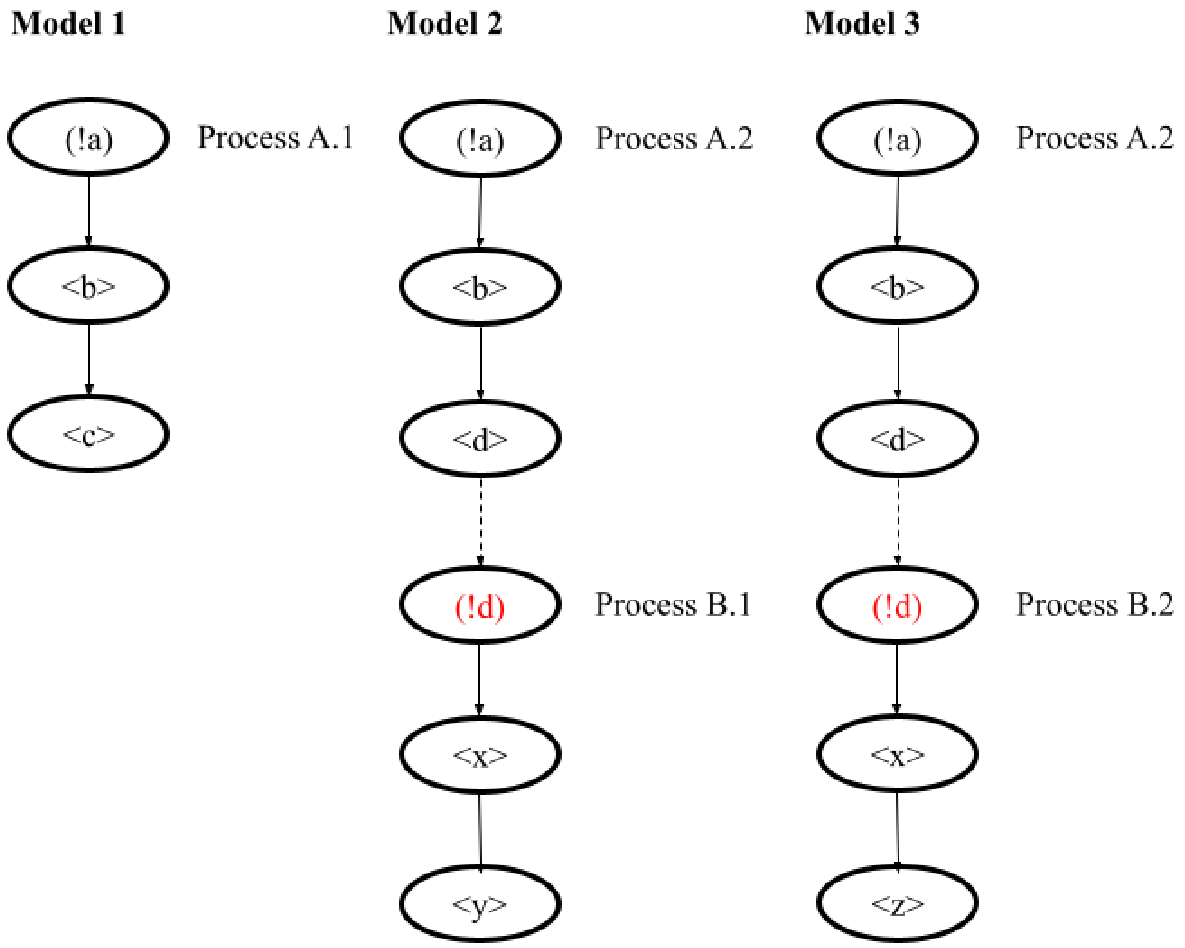
The second tactic (*linear*) splits process actions into sequences of actions without choices. Together created scenarios cover the exact behavior of the original process.

The example of a model provided above can be split into three models assuming there are no savepoints. A process can be split into two versions: Process *A.1* and Process *A.2*. The first model does not contain any versions of B process, since there is no any activating signal sending to it. Models 2 and 3 have *A.2* process and *B.1* and *B.2* correspondingly.

The third tactic (*savepoint_requirements*) splits processes into scenarios reducing the number of branches in choices. It extracts savepoint requirements and looks into lists of provided actions. These actions are used to select first actions of required branches in choices. The tactic starts from the first process's action and traverses actions moving to leaf actions. Thus, the order of actions in requirements matters, as it influences processing of choices. The tactic generates much fewer scenarios than the linear one because it reduces branches only for those choices for which there are provided actions in requirements. Moreover, it does not generate scenarios for branches that are excluded by requirements. The tactic generates specific scenarios for each requirement with provided actions for each process given in every savepoint requirement. So a process may have several scenarios generated for certain savepoints exclusively.

The next step of decomposition of an IEM is scenario selection. It means that the origin IEM is copied, then each process is replaced by a scenario prepared from it. Some processes can be deleted because they cannot be activated anymore or they are unnecessary after adding a scenario with a savepoint to the new IEM.

There are several attributes in processes that influence the whole model: declarations and definitions. For the sake of comfortable use of these attributes, the EMG tool keeps declarations and definitions even from processes that are excluded from generated models.


Model 1
Model 2
Model 3
(!a)
Process A.1
<b>
<c>
(!a)
Process A.2
<b>
<d>
(!d)
Process B.1
<x>
<y>
(!a)
Process A.2
<b>
<d>
(!d)
Process B.2
<x>
<z>

There are several implementations of scenario selection. The *select scenarios* configuration property allows choosing a tactic for scenario selection. There are the following tactics with the corresponding configuration property values in parentheses:

- **Default** (None) – the tactic only adds an extra environment model to the original one per each found savepoint.
- **Combinatorial** (*use all scenarios combinations*) – the tactic generates all possible combinations of scenarios in environment models filtering out infeasible ones.
- **Selective** (a dictionary with configuration is given) – the tactic allows users to set which particular scenarios should be added to new environment models.
- **Savepoints-based** (*select savepoints*) – the tactic allows users to choose scenarios generated for savepoints. It is intended to use this tactic with the *savepoint_requirements* process separation tactic.

To activate decomposition, one should set the *single environment model per fragment* configuration property to True. There are additional configuration parameters to manage the decomposition listed below:

Table 1.13: Decomposition configuration parameters.

Configuration Parameter	Value Type	Default Value	Description
scenario separation	None, linear, savepoint_requirements	None	Allows choosing a process separation tactic.
select scenarios	*use all scenarios combinations*, *select savepoints*, Obj or None	None	Allows to select one of listed above scenario selection tactics.
skip origin model	bool	False	Skip the provided original IEM and do not provide it together with new generated models.
skip savepoints	bool	False	It is relevant for default and combinatorial factories to generate models. If the flag is set, then no extra models with savepoint scenarios will be outputted.
savepoints	A map from process identifiers to either list of savepoint names or bool flag. True value means that all savepoints can be included. False means no savepoints should be included. A list just enumerates savepoints to include. Omitted processes are ignored. {“c1/p1”: true, “c1/p2”: [“sp1”]}	False	The configuration parameter is intended only for the *savepoints-based* tactic. It allows to filter savepoints for which models should be prepared.

The *selective* tactic allows a user to select scenarios for IEMs for each process. Names of scenarios are generated automatically, so they are assumed to be unknown to users. Thus, it is possible to implicitly include them by providing savepoint names and actions that should be selected for output models.

Complicated models can be decomposed in many scenarios, so there could be even more combinations of scenarios. There are three kinds of configuration parameters to restrict the number of environment models in output. They are given below.

Table 1.14: Configuration parameters of the selective tactic.

Configuration Parameter	Value Type	Default Value	Description
must contain	Map from process identifiers to must contain selection descriptions for the property: {“category/name”: {. . . }}	{}	The value lists processes that must be in every generated after decomposition environment model.
must not contain	Map from process identifiers to must contain selection descriptions for the property: {“category/name”: {. . . }}	{}	The value lists processes that must be removed from every generated after decomposition environment model.
cover scenarios	Map from process identifiers to coverage descriptions for the property: {“category/name”: {. . . }}	The parameter is necessary and should not be empty.	Names enumerate process identifiers with actions and savepoints covered by at least a single generated IEM.

The *must contain* configuration property allows a user to select actions and savepoints that must be in any environment model. There are attributes of selection descriptions for the *must contain* configuration property provided below.

Table 1.15: Members of “must contain” configuration parameter.

Configuration Parameter	Value Type	Default Value	Description
actions	List of lists of action names. Example: [[“a”, “b”], [“c”, “d”]]	[]	The list contains corteges of actions that should be in the process in each environment model. If several corteges are provided, then an output model may have all actions of any cortege in the corresponding selected scenario.
savepoint	A name of the savepoint that should be added to all output environment models.	None	If the attribute is set, then each output model will contain a scenario with the provided savepoint.
scenarios only	Bool	True	If the attribute is set to True, then scenarios of a process except the original process can be selected to the environment model.

Suppose the description is an empty object or has only the *scenarios only* flag set. In that case, it is assumed that the output environment model should contain at least one scenario for the process or the original process itself (*scenarios only* is set to False).

There are attributes of selection descriptions for the *must not contain* configuration property that works oppositely to the *must contain* parameter.

Table 1.16: Members of “must not contain” configuration parameter.

Configuration Parameter	Value Type	Default Value	Description
actions	List of actions. [“a”, “b”]	[]	Output models will not have any actions listed in the attribute value.
savepoints	List of savepoint names. [“a”, “b”]	[]	Output models will not have any savepoints listed in the attribute value.

The *cover scenarios* parameter is always necessary. It lists processes and their actions and savepoints that should be covered by at least one environment model in the output of the decomposition step. There are the following attributes to configure the description for a process:

Table 1.17: Members of “cover scenarios” configuration parameter.

Configuration Parameter	Value Type	Default Value	Description
actions	A list of action names. [“a”, “b”]	If it is missing, then all actions should be covered.	The list of actions that should be added to at least one output model.
actions except	A list of action names. [“a”, “b”]	Ignored if it is missing.	The value is the list of actions that are removed from the list of actions that should be covered. Note that provided actions can be added to output models but not ought to be. If almost all actions should be covered, it is helpful to set the property instead of the *actions* one.
savepoints	A list of savepoint names. [“a”, “b”]	If it is missing, then all savepoints should be covered.	The list of savepoints that should be added to at least one output model.
savepoints except	A list of savepoint names. [“a”, “b”]	Ignored if it is missing.	The value is the list of savepoints that are removed from the list of savepoints that should be covered. Note that provided savepoints can be added to output models but not ought to be. If almost all savepoints should be covered, it is helpful to set the property instead of the *savepoints* one.

The selective strategy tries to reduce the number of output IEMs. It resolves dependencies between processes and scenarios, and for each action and savepoint generates at least one model. However, the output set of models can still be quite large, and some actions or savepoints may be selected several times, or generated models can contain actions that are not marked for coverage. If the output model does not include what you want, check configuration properties and signal dependencies between processes because provided configurations can be infeasible.

1.7.5 Example of Specification Decomposition

Let's go through the main modeling steps to prepare a manual model for a Linux device driver. We highly recommend everybody who wants to apply Klever to his/her software. Modeling for Linux device drivers does not require writing specifications from scratch but allows practice in many steps of modeling.

Prepare the UDEMS

Note: It works only for the development installation of Klever.

Klever's installation has several examples to try. One of those is the *Loadable kernel modules sample* preset. Let us just simplify the `job.json` of this sample a bit and start verification:

```
{
  "project": "Linux",
  "build base": "linux/loadable kernel modules sample",
  "targets": ["drivers/ata/pata_arasan_cf.ko"],
  "specifications set": "3.14",
  "requirement specifications": ["empty"]
}
```

The job description forces Klever to run verification of the *drivers/ata/pata_arasan_cf.ko* driver against an *empty* rule. The empty rule does not check anything but it allows to estimate the coverage of the source code roughly and check that the model generation works well. The check of the empty rule is the fastest possible. The Klever should report the *Safe* verdict.

Then you can either open the appropriate EMG page in the web UI using the following sequence of components *Core → Job → VTG → EMGW → EMG*, get the prepared in advance, complete content of UDEMS from that page and put it to, say, `presets/jobs/specifications/linux/pata user model.json` in *$KLEVER_SRC* or do this in the way described below.

You can go to the installation directory of the Klever and copy file `klever-core-work-dir/job/vtg/drivers/ata/pata_arasan_cf.ko/empty/emg/0/input model.json` in `klever-work/native-scheduler/scheduler/jobs/<job ID>/` of *$KLEVER_DEPLOY_DIR* with an IEM to the directory with Klever specifications `presets/jobs/specifications/linux` in *$KLEVER_SRC*. The model should be correct. Just add framing members as the format of UDEMS requires. Note, that the file should be renamed by adding a user model suffix to it. Let us name the file `pata user model.json`. The file should look like this where *3.14* is the name of the specifications set:

```
{
  "3.14": [
    {
      "fragments": [
        "drivers/ata/pata_arasan_cf.ko"
      ],
      "model": {}
    }
  ]
}
```

Then we have to change options of the EMG to run only genericManual generator to prepare our model. Find the proper template in the `Linux.json` file (it is the first one that contains the EMG value) and fix the configuration parameters of EMG as follows:

```
{
  "templates": {
    "loadable kernel modules and kernel subsystems": {
      "plugins": [
        {
          "name": "EMG",
          "options": {
            "generators options": [
              {"genericManual": {}}
            ],
            "translation options": {
              "allocate external": false
            }
          }
        }
      ]
    }
  }
}
```

Run Klever again with new configuration parameters and UDEMS. The expected result is Safe again.

Generated models are not tidy enough. We can simplify them by doing the following transformations:

1. Check the source code of the driver. We can see that the PM-related scenario has many callbacks which are not implemented. Let us keep only the suspend-resume pair.

1. Remove all actions except *pm_deregister*, *pm_register*, *sus*, *suspend_34*, *post_call_33*, *sus_ok*, *sus_bad*, *res*, *resume_22*, *post_call_21*.
2. Rename actions with suffixes to get rid of numerical suffixes. Move the code from post_call actions to suspending and resuming actions and delete formers. Rename *sus_ok* to *ok* and do the same with other ok/bad actions.
3. Then remove jumping actions, there are too many of them. Use *normal*, *sus*, *res* subprocesses to make a new actions sequence without loops and checking the return value of resuming callback.

   “process”: “(!pm_register).(<suspend>.(<ok>.<resume>|<bad>).(pm_deregister)”
4. Add a call of `ldv_assume()` to the resuming action to make it always expect a successful return value of the callback.
5. Remove the unnecessary *replicative* member from the *pm_deregister* action.
6. Remove unused label *pm_ops*.

2. Next, it is time to clean up the *platform_instance_arasan_cf_driver* process.

1. Merge *pre_call_0*, *probe_2* and *post_call_1* actions. Name the final action *probe*. Choose shorter names for *positive_probe* and *negative_probe* actions such as *ok* and *bad*.
2. Remove actions intended for calling callbacks by pointers: *pre_call_6*, *suspend_8*, *post_call_7*, *resume_4*, *shutdown_5*.
3. Rename *release_3* to *release*.
4. Move left actions from *call* to *main* to make a sequential order of actions. Remove the *call* action to get process order as in the snippet given below.
5. Remove the unused label *emg_param_1_0*.
6. Remove replicative entry from the dispatch as it is not required.

```
{
  "main": {
    "comment": "Check that device is truly in the system and begin callback invocations.
↪",
    "process": "<probe>.(<ok>.([pm_register].[pm_deregister]|<none>).<release>.<after_
↪release>|<bad>).<free>.(deregister)"
  }
}
```

The descriptions of processes will be looking as follows (we used formatting to make the text as shorter as possible):

```
{
  "platform/platform_instance_arasan_cf_driver": {
    "actions": {
      "after_release": {
        "comment": "Platform device is released now.",
        "statements": [
          "%probed% = 1;"
        ]
      },
      "deregister": {
        "comment": "Finish {} callbacks invocations scenario.",
        "condition": [
          "%container% == $ARG1",
          "$ARG1 == emg_alias_arasan_cf_driver"
        ],
        "parameters": [
          "%container%"
        ],
        "trace relevant": true
      },
      "free": {
        "comment": "Free memory for 'platform_device' structure.",
        "statements": [
          "$FREE(%resource%);"
        ]
      },
      "init": {
        "comment": "Alloc memory for 'platform_device' structure.",
        "statements": [
          "$ALLOC(%resource%);",
          "%resource%->id_entry = & %container%->id_table[0];"
        ]
      },
      "main": {
        "comment": "Check that device is truely in the system and begin callback␣
↪invocations.",
        "process": "<probe>.(<ok>.([pm_register].[pm_deregister]|<none>).<release>.
↪<after_release>|<bad>).<free>.(deregister)"
      },
      "bad": {
        "comment": "Failed to probe the device.",
        "condition": [
```

(continues on next page)

(continued from previous page)

```
          "%probed% != 0"
        ]
      },
      "none": {
        "comment": "Skip callbacks call."
      },
      "pm_deregister": {
        "comment": "Finish the power management scenario.",
        "parameters": []
      },
      "pm_register": {
        "comment": "Proceed to a power management scenario.",
        "parameters": []
      },
      "ok": {
        "comment": "Platform device is probed successfully now.",
        "condition": [
          "%probed% == 0"
        ]
      },
      "probe": {
        "comment": "Check that the device in the system and do driver initializations.",
        "statements": [
          "ldv_pre_probe();",
          "%probed% = emg_wrapper_arasan_cf_probe(%resource%);",
          "%probed% = ldv_post_probe(%probed%);"
        ],
        "trace relevant": true
      },
      "register": {
        "comment": "Register a driver callbacks for platform-level device.",
        "condition": [
          "$ARG1 == emg_alias_arasan_cf_driver"
        ],
        "parameters": [
          "%container%"
        ],
        "trace relevant": true
      },
      "release": {
        "comment": "Remove device from the system.",
        "statements": [
          "emg_wrapper_arasan_cf_remove(%resource%);"
        ],
        "trace relevant": true
      }
    },
    "category": "platform",
    "comment": "Invoke platform callbacks. (Relevant to 'arasan_cf_driver')",
    "declarations": {
      "environment model": {
        "emg_wrapper_arasan_cf_probe": "extern int emg_wrapper_arasan_cf_probe(struct␣
↪platform_device *);\n",
```

(continues on next page)

(continued from previous page)

```
        "emg_wrapper_arasan_cf_remove": "extern int emg_wrapper_arasan_cf_remove(struct␣
↪platform_device *);\n"
      }
    },
    "definitions": {
      "/var/lib/klever/workspace/Branches-and-Tags-Processing/linux-stable/drivers/ata/
↪pata_arasan_cf.c": {
        "emg_wrapper_arasan_cf_probe": [
          "/* EMG_WRAPPER emg_wrapper_arasan_cf_probe */\n",
          "int emg_wrapper_arasan_cf_probe(struct platform_device *arg0) {\n",
          "\treturn arasan_cf_probe(arg0);\n",
          "}\n",
          "\n"
        ],
        "emg_wrapper_arasan_cf_remove": [
          "/* EMG_WRAPPER emg_wrapper_arasan_cf_remove */\n",
          "int emg_wrapper_arasan_cf_remove(struct platform_device *arg0) {\n",
          "\treturn arasan_cf_remove(arg0);\n",
          "}\n",
          "\n"
        ]
      }
    },
    "headers": [
      "linux/mod_devicetable.h",
      "linux/platform_device.h"
    ],
    "labels": {
      "container": {
        "declaration": "struct platform_driver *container",
        "value": "emg_alias_arasan_cf_driver"
      },
      "probed": {
        "declaration": "int probed",
        "value": "1"
      },
      "resource": {
        "declaration": "struct platform_device *resource"
      }
    },
    "peers": {
      "functions models/__platform_driver_register": [
        "register"
      ],
      "functions models/platform_driver_unregister": [
        "deregister"
      ],
      "pm/pm_ops_scenario_arasan_cf_pm_ops": [
        "pm_deregister",
        "pm_register"
      ]
    },
```

(continues on next page)

(continued from previous page)

```
    "process": "(!register).<init>.{main}"
  },
  "pm/pm_ops_scenario_arasan_cf_pm_ops": {
    "actions": {
      "pm_deregister": {
        "comment": "Do not expect power management scenarios.",
        "parameters": [],
        "trace relevant": true
      },
      "pm_register": {
        "comment": "Ready for a power management scenarios.",
        "parameters": [],
        "trace relevant": true
      },
      "resume": {
        "comment": "Make the device start working again after resume.",
        "statements": [
          "%ret% = emg_wrapper_arasan_cf_resume(%device%);",
          "ldv_assume(%ret% = 0);"
        ],
        "trace relevant": true
      },
      "bad": {
        "comment": "Callback failed.",
        "condition": [
          "%ret% != 0"
        ]
      },
      "ok": {
        "comment": "Callback successfully finished.",
        "condition": [
          "%ret% == 0"
        ]
      },
      "suspend": {
        "comment": "Quiesce subsystem-level device before suspend.",
        "statements": [
          "%ret% = emg_wrapper_arasan_cf_suspend(%device%);",
          "%ret% = ldv_post_probe(%ret%);"
        ],
        "trace relevant": true
      }
    },
    "category": "pm",
    "comment": "Invoke power management callbacks. (Relevant to 'arasan_cf_pm_ops')",
    "declarations": {
      "/var/lib/klever/workspace/Branches-and-Tags-Processing/linux-stable/drivers/ata/
↪pata_arasan_cf.c": {
        "emg_alias_arasan_cf_pm_ops": "struct dev_pm_ops *emg_alias_arasan_cf_pm_ops = &␣
↪arasan_cf_pm_ops;\n"
      },
      "environment model": {
```

(continues on next page)

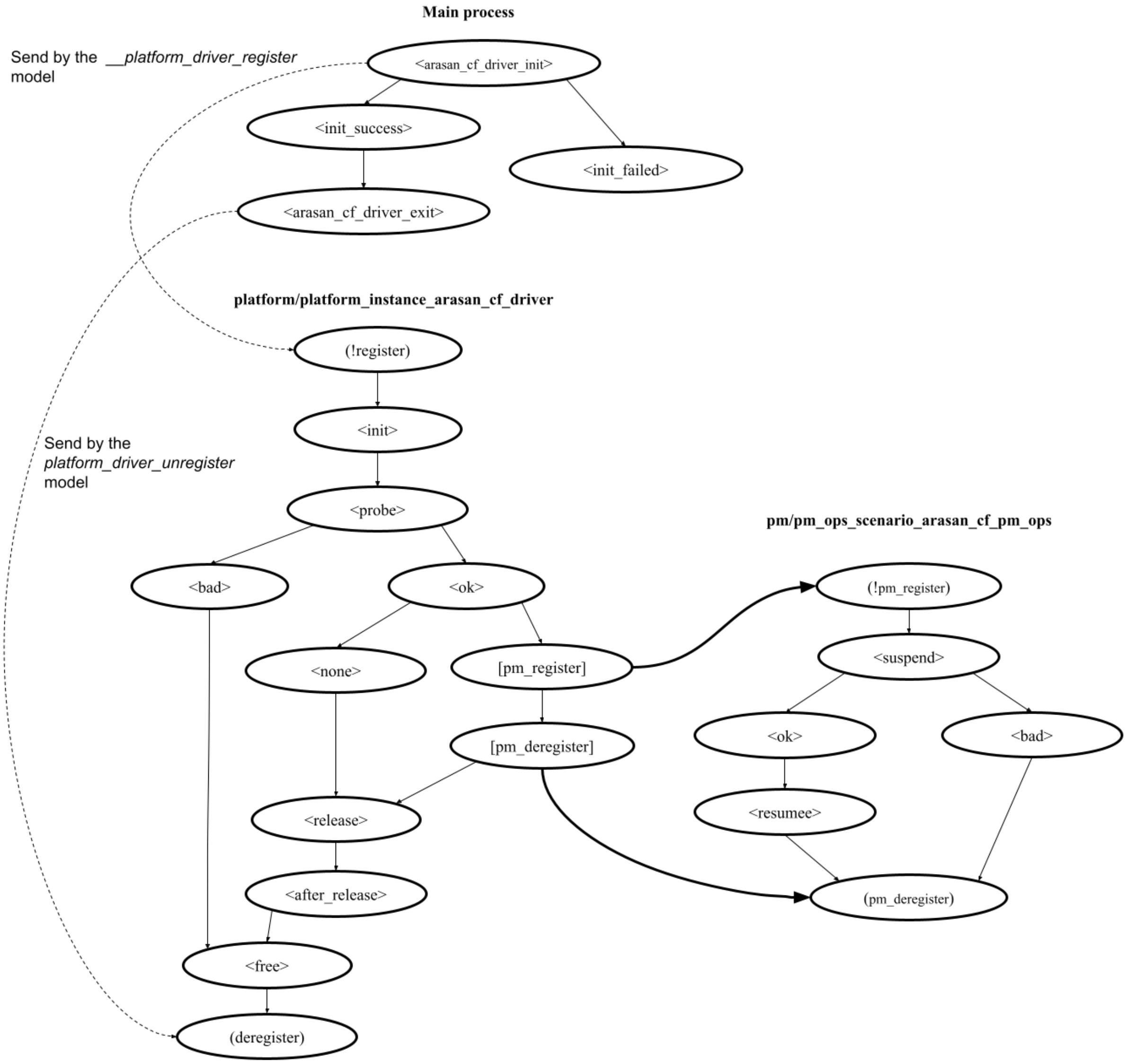
(continued from previous page)

```
        "emg_alias_arasan_cf_pm_ops": "extern struct dev_pm_ops *emg_alias_arasan_cf_pm_
↪ops;\n",
        "emg_runtime_enabled": "int emg_runtime_enabled = 0;\n",
        "emg_runtime_status": "int emg_runtime_lowpower = 1;\n",
        "emg_wrapper_arasan_cf_resume": "extern int emg_wrapper_arasan_cf_resume(struct␣
↪device *);\n",
        "emg_wrapper_arasan_cf_suspend": "extern int emg_wrapper_arasan_cf_
↪suspend(struct device *);\n"
      }
    },
    "definitions": {
      "/var/lib/klever/workspace/Branches-and-Tags-Processing/linux-stable/drivers/ata/
↪pata_arasan_cf.c": {
        "emg_wrapper_arasan_cf_resume": [
          "/* EMG_WRAPPER emg_wrapper_arasan_cf_resume */\n",
          "int emg_wrapper_arasan_cf_resume(struct device *arg0) {\n",
          "\treturn arasan_cf_resume(arg0);\n",
          "}\n",
          "\n"
        ],
        "emg_wrapper_arasan_cf_suspend": [
          "/* EMG_WRAPPER emg_wrapper_arasan_cf_suspend */\n",
          "int emg_wrapper_arasan_cf_suspend(struct device *arg0) {\n",
          "\treturn arasan_cf_suspend(arg0);\n",
          "}\n",
          "\n"
        ]
      }
    },
    "headers": [
      "linux/device.h",
      "linux/pm.h"
    ],
    "labels": {
      "device": {
        "declaration": "struct device *device"
      },
      "ret": {
        "declaration": "int ret",
        "value": "ldv_undef_int()"
      }
    },
    "peers": {
      "platform/platform_instance_arasan_cf_driver": [
        "pm_deregister",
        "pm_register"
      ]
    },
    "process": "(!pm_register).(<suspend>.(<ok>.<resume>|<bad>)).(pm_deregister)"
  }
}
```

Now, a user can add his/her own extensions to these models. Function models' descriptions we have left as is.

Rename actions *init_failed_0* and *init_success_0* to *init_failed* and *init_success* in the main process correspondingly.

There are environment processes and the main process of the generated environment model in the picture below. There are three processes. The main process starts doing its actions first. Then it registers the *platform/platform_instance_arasan_cf_driver* process implicitly by function models called at the initialization function. The deregistration of the process is also implicit. Dashed arrows visualize possible signals. The last-mentioned process can register *pm/pm_ops_scenario_arasan_cf_pm_ops* in case of the successful probe. These arrows have a bold style.



Decompose the UDFS

Let us consider several examples of decomposition of the model provided above.

The first step is adding savepoints. If the driver would be complicated, then we did add savepoints to environment model processes. But it is rather simple in our case. That is why we consider the more interesting case: how to implement several versions of the model using savepoints.

To keep the model as is but add several savepoints, it is required to add them to the main process. But it is difficult to use them alongside with the selective tactic that we are going to use for this example. The solution is to move the process to the *environment processes*:

1. Add a new *main/process* member to *environment processes*.
2. Move the process description to this new entry.
3. Set *main process* to *null*.
4. Leave its parameters empty. This action is necessary to correspond to the requirement that all environment processes must have an activating receiving action.

Note, that there is no main process any more. Such a model cannot be provided without either linuxInsmod generator added to the environment generator pipeline or activated decomposition. We would like to choose the latter case.

The main process does not have peers. But it calls the initialization function that calls the platform registering function in turn. We need to specify this dependency as it is implicit for the EMG. Add the following member to the *register* action of the *platform/platform_instance_arasan_cf_driver* process. It allows us to reduce the number of models generated at decomposition by fulfilling this requirement.

```
"require": {
  "processes": {"main/process": true},
  "actions": {"main/process": ["init_success"]}
}
```

Finally, we can add a savepoint to the *main_register* action of *main/process*.

```
"savepoints": {
  "demo": {
    "comment": "The savepoint added for demonstrating purposes.",
    "statements": []
  }
}
```

Next we can run the model. One needs to activate decomposition and select the proper selection tactics. We are going to separate the model into simpler scenarios. It is useful to use linear scenarios in this case. All variants of action sequences will be split in separate scenarios. But it will result in many scenario combinations. Thus, we choose the selective tactic for scenario selection to reduce their number.

Set additional configuration properties in `job.json`:

```
{
  "scenario separation": "linear",
  "single environment model per fragment": false
}
```

Finally we consider several versions of configuration and discuss what they result in.

1. **Cover only the failed initialization function**. In the case we need only the main process and the branch of the choice operator with *init_failed* action. Thus, we set this action as a single to cover. The *savepoints only*

parameter forces the Decomposer to generate models only with savepoints of *main/process* scenario. There is a single model should be generated of this configuration:

```
"select scenarios": {
  "cover scenarios": {
    "main/process": {"actions":  ["init_failed"], "savepoints only":  true}
  }
}
```

2. **Cover successful invocation of the probe callback but without suspend-resume callbacks**. The explicit ban of *pm/pm_ops_scenario_arasan_cf_pm_ops* is the main difference of this configuration from the previous one.

```
"select scenarios": {
  "cover scenarios": {
    "platform/platform_instance_arasan_cf_driver": {"actions":  ["ok"]}
  },
  "must not contain": {
    "pm/pm_ops_scenario_arasan_cf_pm_ops": {}
  }
}
```

3. **Cover suspend-resume callbacks without failing initialization and probing callbacks**. In this example we add a requirement that each model must contain a "pm_register" signal sending action.

```
"select scenarios": {
  "cover scenarios": {
    "platform/platform_instance_arasan_cf_driver": {},
    "pm/pm_ops_scenario_arasan_cf_pm_ops": {"actions": ["suspend", "resume"]}
  },
  "must contain": {
    "platform/platform_instance_arasan_cf_driver": {"actions": [["pm_register"]]}
  }
}
```

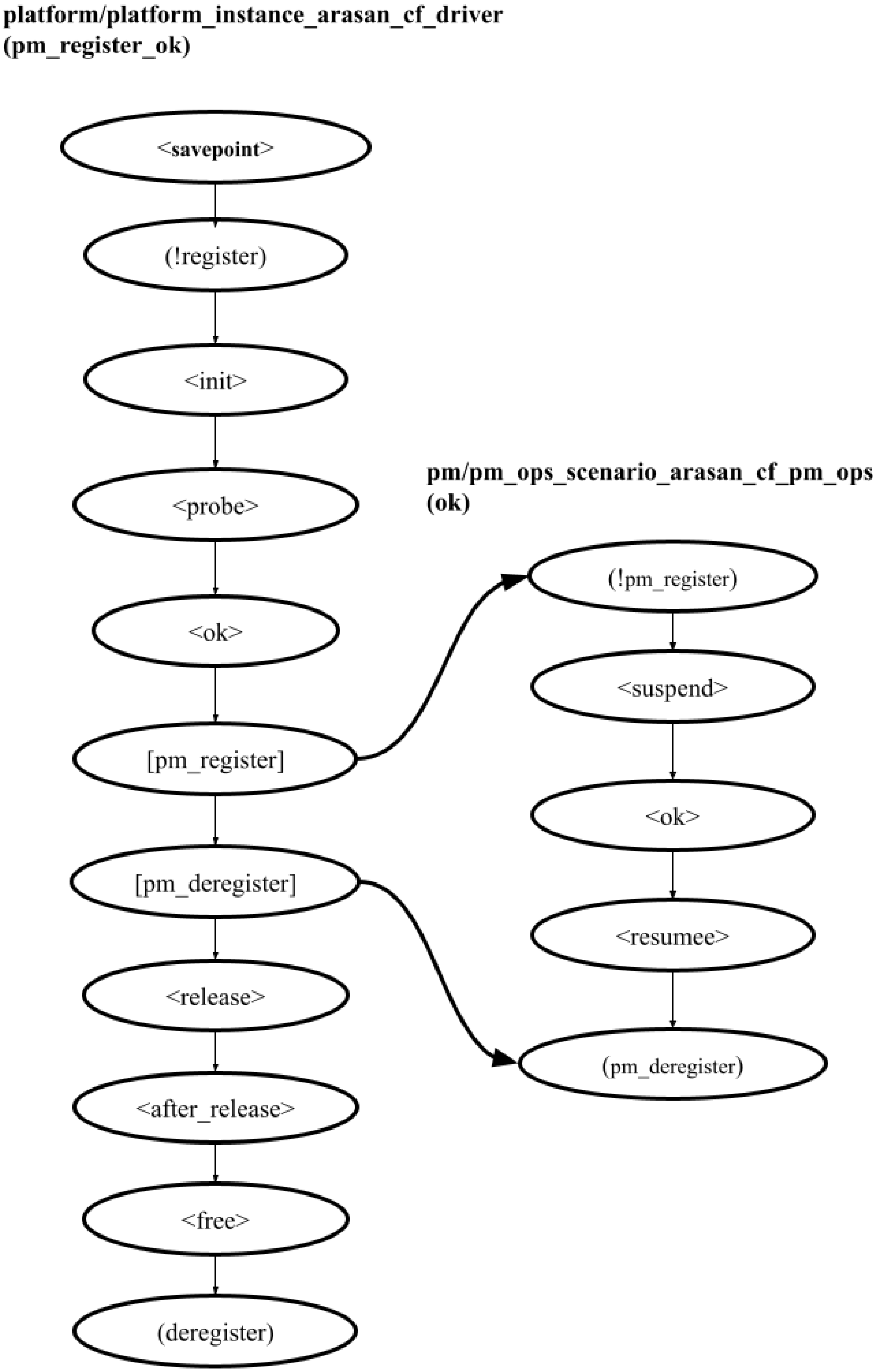
Note, that if the result of decomposition is unexpected to you, then you need to state more explicit options. Previous examples did not contain all requirements of actions and not all processes were mentioned also. But it is so because of implicit dependencies between processes. If you do not understand some of them, then it is easier to specify coverage and contain each process. You may relax after finding a working solution.

The linear tactic splits processes into variants of sequences of actions without any choices. It assigns names to scenarios using the savepoint name any is selected and chosen first actions of the branch of each choice operator. Then, the selective tactic chooses scenarios for each process. One can see the names of chosen scenarios in the web interface.

Let's consider one of the generated environment models. It has:

- the *init_success* scenario for *main/process demo*,
- the *pm_register_ok* for *platform/platform_instance_arasan_cf_driver*,
- the *ok* scenario for *pm/pm_ops_scenario_arasan_cf_pm_ops*.

The picture below shows scenarios of the generated environment model. You can see that there are no choices. Signals are left as is. The *init_success* scenario has the first *demo* base block action generated from the savepoint.


platform/platform_instance_arasan_cf_driver
(pm_register_ok)
<savepoint>
(!register)
<init>
<probe>
<ok>
[pm_register]
[pm_deregister]
<release>
<after_release>
<free>
(deregister)
pm/pm_ops_scenario_arasan_cf_pm_ops
(ok)
(!pm_register)
<suspend>
<ok>
<resumee>
(pm_deregister)

1.7.6 Environment Model Translator

The *model translator* translates an input IEM into parallel or sequential C code according to the configuration parameters, checked safety property, and used a verification tool.

If it is used with the decomposer, then each IEM is translated independently from others.

The result of translation is not semantically equivalent to the IEM. There is an approach to do that theoretically correct, but it does not make sense in practice. State-of-the-art verification tools cannot verify large multi-threaded programs against reachability and memory safety properties. Thus, translation includes simplifications and restrictions on an input model to guarantee the plainness of obtained C code.

Checking reachability and memory safety requires a sequential environment model. The model translator component accepts only processes with a single signal receive that is lso always a first action and optional receives as last actions. There are also dispatches allowed. The translator prepares a *control function* for the process that can be called in places of sending signals to the translated process by others.

Interleavings of actions of different processes are not implemented to simplify the FEM. Memory safety configuration preset differs only by assignment of a specific function call to all pointer values.

The parallel model for data race detection is multithreaded and does not support signal receiving other than the first and terminal positions in the process's transfer relation. We do not consider it there for simplicity.

Table 1.18: Configuration parameters of Translator.

Configuration Parameter	Value Type	Default Value	Description
entry point	Function name string	main	The value is the name of the environment model entry point function.
environment	String	environment_model.c	The name of the main C file.
direct control functions calls	Bool	True	Call control functions directly to make the sequential model or create/join pthreads at signal sending.
simple control functions calls	Bool	True	Make control function calls simpler a bit by removing memory allocation for parameters. Turn it off for a parallel model generation.
code additional aspects	List of strings	[]	The parameter allows adding additional aspect files to the environment model output files. Paths are relative to the directory with specifications.
additional header	List of strings	["ldv/linux/common.h", "ldv/linux/err.h", "ldv/verifier/common.h", "ldv/verifier/gcc.h", "ldv/verifier/nondet.h", "ldv/verifier/memory.h", "ldv/verifier/thread.h"]	The parameter allows including additional header files to the main C file. Paths are relative to the directory with specifications.
propagate headers to instrumented files	Bool	True	Header files provided with the help of the previous configuration options become included in the program fragment files.
self parallel processes	Bool	False	The generated parallel model creates several threads per a control function.
not self parallel processes	List of strings	[]	A list of process identifiers for which the translator creates a single thread in the generated parallel model despite the previous active configuration parameter.
ignore missing function models		False	Do not generate models of functions, if they are not found in the program fragment file and the configuration parameter is valid.
implicit signal peers		False	If the configuration parameter is valid, then the translator attempts to find peers by matching signals. The option is needed if the provided IEM misses attributes that describe peers.
do not skip signals		False	If the parameter is set, the translator removes signal dispatches and receives for actions for which there are no known peers.
initialize		True	The translator inserts an initialization function (ldv_initialize) at the beginning of the environment

1.8 Development of Verifier Profiles

Verification tools have different capabilities. Klever extensively uses CPAchecker as a verification backend. CPAchecker has a variety of analyses implemented. Klever primarily uses predicate analysis, value analysis, symbolic memory graphs and finding data races. A user may need to know how to configure Klever to apply the best suiting configuration of a verification tool to cope with a particular program. The choice would depend on a program complexity and safety property to check the program against. Function pointers management, dynamic data structures, arrays, floating-point and bit-precise arithmetics, parallel execution are supported only by specific configurations. Specific safety properties such as memory safety or the absence of data races can be checked only by particular analyses. Klever provides a format to define configurations of specific safety properties that can be chosen or modified manually to adjust them for user needs. Such configuration files are called *verification profiles* and this section gives a tutorial on them.

Verification profiles are described in the `presets/jobs/verifier profiles.json` file in the *$KLEVER_SRC* directory. Below we consider its structure. The first attribute of the file is *templates*. Each template can have:

- A comment (*description*).
- Options set by *add options* and *architecture dependant options*
- The ancestor defined by *inherit*.

Options are collected from the ancestor and then updated according to options described in *add options* and *architecture dependant options*. Inheritance is possible only from another template.

Architecture can be set by the *architecture* attribute in the `job.json` file. It results in choosing the suitable additional options for verification tools provided within *architecture dependant options* entry.

There is an example below with three described options:

```
"add options": [
  {"-setprop": "cpa.callstack.unsupportedFunctions=__VERIFIER_nonexisting_dummy_function
↪"},
  {"-ldv-bam": ""}
  {"-heap": "%ldv:memory size:0.8:MB%m"}
]
```

Each entry is an object that has a single entry providing an option name and its value. Some options have empty strings as values. The last option in the list has a specific format that allows Klever to adjust the amount of memory for the heap. It equals to 80% of memory allocated for the verification tool in the example. You can find a complete list of CPAchecker's options here.

Requirement specifications should be provided with certain verification profiles and verification tools versions. They are described in the *profiles* attribute. Let's consider an example of a *reachability* verification profile:

```
"profiles": {
  "reachability": {
    "CPAchecker": {"klever_fixes:38771": {"inherit": "CPAchecker BAM reachability"}},
    "UltimateAutomizer": {"v0.1.20": {"inherit": "Ultimate common"}}
  }
}
```

The *reachability* profile can be used at description of requirement specifications within appropriate requirement specification bases. The verification profile allows using two verification tools named CPAchecker and UltimateAutomizer. The next objects in the example have verification tools' versions as keys and inheritance descriptions as values. The latter refers the template that is inherited by the verification profile. Note that it is possible to use different templates for distinguished versions of a verification tool in a single verification profile.

You need to refer the developer documentation or contact Klever's developers to learn how to support more verification tools as well as additional configuration options for supported verification tools.

1.9 Verifying New Program

This tutorial shows how one can verify a new program with help of Klever. As an example we consider Simple OS developed for the sake of the given tutorial. Simple OS is an artificial operating system including a kernel and a set of modules that are compiled and linked together into executable file `simple-os`. Their source code is located at directory *$KLEVER_SRC*`/docs/samples/simple-os`. You are able to modify any existing source files and scripts as well as to add new ones if you want to check something in addition. We will be glad to accept any interesting contribution.

You should examine common *Tutorial* beforehand since we will not duplicate its content here. Besides, it would be great if you will be familiar at least to some extent with other topics like *Configuring Program Decomposition*, *Development of Common API Models* and *Development of Requirement Specifications*. Everything below should work well for the *local production instance of Klever*, but we recommend installing Klever in the development mode locally, especially if you are going to perform extra experiments.

1.9.1 Preparing Build Base

At the first step you need to prepare a *build base* for Simple OS[1]. For that purpose you should execute the following commands:

```
$ cd $KLEVER_SRC/docs/samples/simple-os
$ clade -c clade.json --cif $KLEVER_DEPLOY_DIR/klever-addons/CIF/bin/cif make -j8
```

The resulting build base will be placed at directory *$KLEVER_SRC*`/docs/samples/simple-os/clade`. To regenerate the build base, e.g. after changing program source files, you should run the following commands before launching Clade:

```
$ make clean && rm -rf clade
```

After successful preparation of the build base that includes the normal build process, you can run Simple OS. It is not necessary for its verification by means of Klever, but this clarifies its workflow and demonstrates possible bugs.

The first command-line argument for `simple-os` specifies a module to be launched:

- *0* is *simple*.
- *1* is *simple-double-allocation* (there is a specially introduced bug).
- *2* is *simple-no-check* (there is a bug).
- *3* is *simple-no-release* (there is a bug).
- *4* is *complex*.

The second command-line argument for `simple-os` provides the only integer argument to a module initialization function.

Here are several examples of launching modules (we encourage you to try more examples, especially if you modify anything in the source code of Simple OS):

[1] The build base for Simple OS prepared in advanced is deployed together with Klever by default. You can refer to it as *simple-os* in `job.json`, but we advise you to prepare the build base yourself.

```
# Module "simple" properly handles all invalid inputs and follows rules of correct usage of the kernel API.
$ ./simple-os 0 2; echo "Kernel exit code is '$?'"
Kernel resource was successfully allocated and initialized.
Kernel resource was successfully released.
Kernel exit code is '0'
$ ./simple-os 0 3; echo "Kernel exit code is '$?'"
Allocation of kernel resource with odd initial values always fails.
Module initialization fails with '-100' error code.
Kernel exit code is '156'
$ ./simple-os 0 10; echo "Kernel exit code is '$?'"
Kernel resource was successfully allocated and initialized.
Kernel resource was successfully released.
Kernel exit code is '0'
$ ./simple-os 0 0; echo "Kernel exit code is '$?'"
Initial value for kernel resource should not be '0'.
Module initialization fails with '-100' error code.
Kernel exit code is '156'

# Module "simple-double-allocation" performs invalid double allocation of kernel resource.
$ ./simple-os 1 2; echo "Kernel exit code is '$?'"
Kernel resource was successfully allocated and initialized.
You should not allocate kernel resource twice.
Kernel resource was successfully released.
Module initialization fails with '-101' error code.
Kernel exit code is '155'
$ ./simple-os 1 3; echo "Kernel exit code is '$?'"
Allocation of kernel resource with odd initial values always fails.
Module initialization fails with '-100' error code.
Kernel exit code is '156'

# Module "simple-no-check" does not check for status of kernel resource allocation.
$ ./simple-os 2 2; echo "Kernel exit code is '$?'"
Kernel resource was successfully allocated and initialized.
Kernel resource was successfully released.
Kernel exit code is '0'
$ ./simple-os 2 3; echo "Kernel exit code is '$?'"
Allocation of kernel resource with odd initial values always fails.
Segmentation fault (core dumped)
Kernel exit code is '139'

# Module "simple-no-release" forgets to release allocated kernel resource in some cases.
$ ./simple-os 3 2; echo "Kernel exit code is '$?'"
Kernel resource was successfully allocated and initialized.
Kernel resource was successfully released.
Kernel exit code is '0'
$ ./simple-os 3 3; echo "Kernel exit code is '$?'"
Allocation of kernel resource with odd initial values always fails.
Module initialization fails with '-100' error code.
Kernel exit code is '156'
$ ./simple-os 3 10; echo "Kernel exit code is '$?'"
Kernel resource was successfully allocated and initialized.
```

(continues on next page)

(continued from previous page)

```
Module allocated kernel resource, but it did not release it.
Kernel exit code is '246'

# Module "complex" is correct as module "simple", but it consists of several source files.
$ ./simple-os 4 2; echo "Kernel exit code is '$?'"
Kernel resource was successfully allocated and initialized.
Kernel resource was successfully released.
Kernel exit code is '0'
$ ./simple-os 4 3; echo "Kernel exit code is '$?'"
Allocation of kernel resource with odd initial values always fails.
Module initialization fails with '-101' error code.
Kernel exit code is '155'
```

Build processes can be very project specific. We can only recommend getting a successful standalone build first and then to wrap it with Clade that should also finish successfully. You can build the whole project even though you are going to verify only its subset. For the rest you should investigate the project documentation, contact developers or be one of them.

1.9.2 Developing Project Specific Adaptation

Klever should be taught so that it will be aware what parts of your project should be verified and what should be checked for them. The appropriate stuff in general is referred to as a *project specific adaptation*. This section will demonstrate the project specific adaptation for Simple OS. It is worth noting, that we are going to verify only separate modules of Simple OS.

We include the project specific adaptation for Simple OS into the repository to simplify the tutorial. Nevertheless, you can track that it corresponds to the following description and even recreate it from scratch similarly but with slightly different names and identifiers.

Describing new preset verification job

First of all you should describe a new preset verification job. For this it is necessary to create directory *$KLEVER_SRC*/presets/jobs/simpleos/modules and put files job.json and tasks.json there.

job.json may look as follows (you can find some details about specified attributes in *Starting Verification*):

```
{
  "project": "SimpleOS",
  "build base": "$KLEVER_SRC/docs/samples/simple-os/clade",
  "targets ": ["**"],
  "specifications set": "1.0",
  "requirement specifications": [".*"]
}
```

Note: You should replace $KLEVER_SRC with an actual path above.

You can copy tasks.json from, say, *$KLEVER_SRC*/presets/jobs/linux/loadable kernel modules sample/tasks.json. The given tutorial does not consider attributes from that file as well as any modifications of them.

Then you need to refer to the new preset verification job in the verification job base. For that goal you can add the following description at the end of file *$KLEVER_SRC*/presets/jobs/base.json:

```
{
  "name": "Simple OS",
  "uuid": "42963a02-abb4-4bc7-a7d1-cdfa1d16d072",
  "production": true,
  "children": [
    {
      "uuid": "fa433698-80e9-4076-b95b-2dad8cbe7e6c",
      "name": "Modules",
      "directory": "simpleos/modules",
      "production": true
    }
  ]
}
```

The verification job base represents a set of trees where:

- Roots correspond to different projects.
- Intermediate nodes can help to arrange a lot of various preset verification jobs, e.g. you can see on a tree for the Linux kernel.
- Leaves correspond to preset verification jobs themselves. Only leaves can have attribute *directory*, and you can create actual verification jobs only on the base of leaves.

These trees are reflected in the Klever web interface, e.g. Fig. 1.2. Attributes in the verification job base have the following meaning:

- *name* - a user-friendly name of a preset verification job, e.g. “Modules” in the example above, or its parents, e.g. “Simple OS”.
- *uuid* - a universally unique identifier. It should be unique across the verification job base. You can take any existing UUID from the verification job base, modify it slightly and use for new preset verification jobs as well as their parents.
- *production* - whether a preset verification job and its parents should be populated for the production instance of Klever. We recommend you to use value *true* for this attribute unless you are going to set up any special-purpose preset verification jobs, e.g. verification jobs for testing and validation.
- *directory* - a path to a directory where you put files `job.json` and `tasks.json`. This path should be relative to directory *$KLEVER_SRC*/presets/jobs.

Describing program fragments

Note: You should read *Configuring Program Decomposition* to understand this subsection better.

Fig. 1.49 demonstrates the command graph for Simple OS. You can find it also in file *$KLEVER_SRC*/docs/samples/simple-os/clade/CmdGraph/cmd_graph_with_files.dot.pdf after successful preparation of the build base.

The command graph demonstrates GCC commands (compilation and linkage) that forms final executable file `simple-os` from a set of input program source files. As it was aforementioned, we are focusing on individual modules in the given tutorial. That is why we can describe them manually in file *$KLEVER_SRC*/presets/jobs/fragmentation sets/SimpleOS.json, for instance:

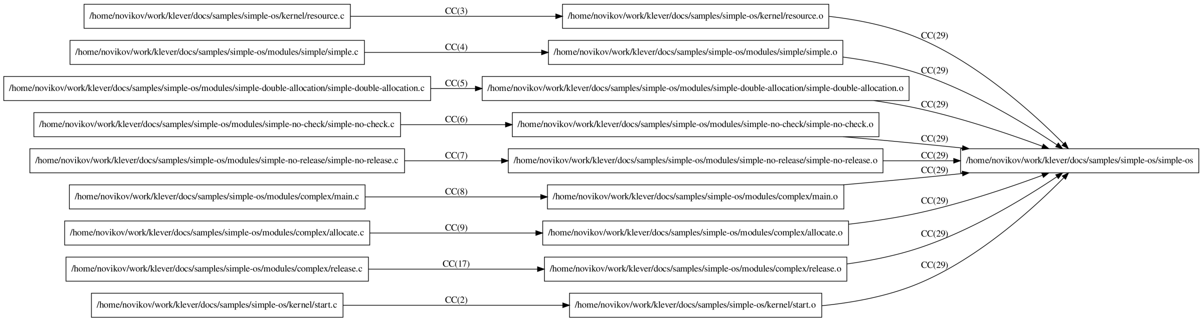

/home/novikov/work/klever/docs/samples/simple-os/kernel/resource.c
CC(3)
/home/novikov/work/klever/docs/samples/simple-os/kernel/resource.o
/home/novikov/work/klever/docs/samples/simple-os/modules/simple/simple.c
CC(4)
/home/novikov/work/klever/docs/samples/simple-os/modules/simple/simple.o
/home/novikov/work/klever/docs/samples/simple-os/modules/simple-double-allocation/simple-double-allocation.c
CC(5)
/home/novikov/work/klever/docs/samples/simple-os/modules/simple-double-allocation/simple-double-allocation.o
/home/novikov/work/klever/docs/samples/simple-os/modules/simple-no-check/simple-no-check.c
CC(6)
/home/novikov/work/klever/docs/samples/simple-os/modules/simple-no-check/simple-no-check.o
/home/novikov/work/klever/docs/samples/simple-os/modules/simple-no-release/simple-no-release.c
CC(7)
/home/novikov/work/klever/docs/samples/simple-os/modules/simple-no-release/simple-no-release.o
/home/novikov/work/klever/docs/samples/simple-os/modules/complex/main.c
CC(8)
/home/novikov/work/klever/docs/samples/simple-os/modules/complex/main.o
/home/novikov/work/klever/docs/samples/simple-os/modules/complex/allocate.c
CC(9)
/home/novikov/work/klever/docs/samples/simple-os/modules/complex/allocate.o
/home/novikov/work/klever/docs/samples/simple-os/modules/complex/release.c
CC(17)
/home/novikov/work/klever/docs/samples/simple-os/modules/complex/release.o
/home/novikov/work/klever/docs/samples/simple-os/kernel/start.c
CC(2)
/home/novikov/work/klever/docs/samples/simple-os/kernel/start.o
CC(29)
CC(29)
CC(29)
CC(29)
CC(29)
CC(29)
CC(29)
CC(29)
CC(29)
/home/novikov/work/klever/docs/samples/simple-os/simple-os

Fig. 1.49: Command graph for Simple OS

```json
{
  "tactics": {"separate modules": {"reference": true}},
  "fragmentation sets": {
    "1.0": {
      "reference": true,
      "fragments": {
        "modules/simple": [
          "modules/simple/simple.c"
        ],
        "modules/simple-double-allocation": [
          "modules/simple-double-allocation/simple-double-allocation.c"
        ],
        "modules/simple-no-check": [
          "modules/simple-no-check/simple-no-check.c"
        ],
        "modules/simple-no-release": [
          "modules/simple-no-release/simple-no-release.c"
        ],
        "modules/complex": [
          "modules/complex/main.c",
          "modules/complex/allocate.c",
          "modules/complex/release.c"
        ]
      },
      "exclude from all fragments": [
        "kernel/resource.c",
        "kernel/start.c"
      ]
    }
  }
}
```

With this configuration we will have 5 program fragments corresponding exactly to modules. Manual description may be a good option in case of stable and simple projects. For large, evolving projects it would be better to develop a script that will automatically construct necessary program fragments and filter out everything irrelevant. You can find examples of these scripts in directory *$KLEVER_SRC*`/klever/core/pfg/fragmentation`.

Describing requirement specifications

Note: We encourage you to read *Development of Common API Models* and *Development of Requirement Specifications* prior to diving into this subsection.

Klever needs custom models and requirement specifications for each new project. Their top level description for Simple OS is resided in requirement specifications base *$KLEVER_SRC*`/presets/jobs/specifications/SimpleOS.json`:

```
{
  "specification sets": ["1.0"],
  "templates": {
    "modules": {
      "plugins": [
        {
          "name": "EMG",
          "options": {
            "generators options": [
              {
                "genericFunctions": {
                  "infinite calls sequence": false,
                  "initialize strings as null terminated": false,
                  "functions to call": ["\\w*init"]
                }
              }
            ],
            "translation options": {
              "entry point": "ldv_main",
              "environment model file": "environment_model.c",
              "additional headers": [
                "ldv/common/model.h",
                "ldv/verifier/common.h",
                "ldv/verifier/nondet.h",
                "ldv/verifier/memory.h"
              ]
            }
          }
        },
        {
          "name": "RSG",
          "options": {
            "common sets model": "counter",
            "model compiler input file": "kernel/start.c",
            "common models": [
              "simpleos/verifier/memory.c",
              "verifier/gcc.c",
              "verifier/nondet.c"
            ]
          }
        },
        {
          "name": "Weaver",
```

(continues on next page)

(continued from previous page)

```
          "options": {"common headers": [
            "ldv/common/inline_asm.h",
            "ldv/verifier/memory.h"
          ]}
        },
        {
          "name": "FVTP",
          "options": {
            "merge source files": true,
            "verifier profile": "reachability",
            "verifier": {
              "name": "CPAchecker",
              "version": "klever_main:3c59e06"
            }
          }
        }
      ]
    }
  },
  "requirement specifications": {
    "description": "Simple OS requirement specifications",
    "template": "modules",
    "children": [
      {
        "identifier": "empty",
        "description": "Get code coverage"
      },
      {
        "identifier": "kernel",
        "children": [
          {
            "identifier": "resource",
            "description": "Check correct usage of kernel resource API",
            "plugins": [{
              "name": "RSG",
              "options": {"models": ["simpleos/kernel/resource.c"]}
            }]
          }
        ]
      },
      {
        "identifier": "memory safety",
        "description": "Check memory safety",
        "plugins": [
          {
            "name": "RSG",
            "options": {"models": ["simpleos/memory safety/memory.c"]}
          },
          {
            "name":"FVTP",
            "options": {
              "verifier profile": "memory checking",
```

(continues on next page)

(continued from previous page)

```
       "verifier": {
        "name": "CPAchecker",
        "version": "klever_main:3c59e06"
       }
      }
     }
    ]
   }
  ]
 }
}
```

The requirement specifications base refers some common models and headers. Besides, most attributes are typical. Below we consider models and settings that are specific for Simple OS.

The value for attribute *functions to call* is *[”\w*init”]*. It means that generated environment models will contain only calls to module initialization functions. In the case of modules of Simple OS it is enough since they have only such entry points.

The value for attribute *model compiler input file* is *kernel/start.c*. Indeed, you can specify any program source file that is compiled during preparation of the build base and that has compilation options suitable for building models.

Model *$KLEVER_SRC*`/presets/jobs/specifications/simpleos/verifier/memory.c` represents models for memory allocation and free:

```
#include <kernel.h>
#include <../verifier/reference memory.c>

void *ldv_malloc(size_t size)
{
    return ldv_reference_malloc(size);
}

void *ldv_xmalloc(size_t size)
{
    return ldv_reference_xmalloc(size);
}

void ldv_free(void *s)
{
    ldv_reference_free(s);
}
```

These models refer corresponding reference models the verification tool familiar with. Simple OS does not have any specific restrictions for dealing with the dynamic memory, so models do not do anything specific.

The requirement specifications base describes 3 requirement specifications some of which refer certain models:

- *empty*
- *kernel:resource*
- *memory safety*

empty is intended for getting code coverage, i.e. code reachable with a given environment model. It does not reveal any issues in target modules of Simple OS.

kernel:resource represents checks for correct usage of the kernel resource API. This requirements specification refers model *$KLEVER_SRC*/presets/jobs/specifications/simpleos/kernel/resource.c:

```
#include <kernel.h>
#include <ldv/common/model.h>
#include <ldv/verifier/common.h>
#include <ldv/verifier/memory.h>

/* There are 2 possible states of kernel resource allocation. */
enum LDV_RESOURCE_ALLOCATION_STATE {
    LDV_NONALLOCATED = 0,   /* Kernel resource is not allocated or it was released. */
    LDV_ALLOCATED           /* Kernel resource is allocated. */
};

static enum LDV_RESOURCE_ALLOCATION_STATE ldv_kernel_resource_state = LDV_NONALLOCATED;

struct resource *ldv_allocate_resource(int init)
{
    struct resource *r;

    if (ldv_kernel_resource_state != LDV_NONALLOCATED)
        /* ASSERT Kernel resource can be allocated only once */
        ldv_assert();

    if (!init || init % 2)
        /* NOTE Could not allocate kernel resource */
        return NULL;

    r = ldv_xmalloc(sizeof(*r));
    r->x = init;

    /* NOTE Allocate kernel resource */
    ldv_kernel_resource_state = LDV_ALLOCATED;

    /* NOTE Kernel resource was successfully allocated */
    return r;
}

void ldv_release_resource(struct resource *r)
{
    if (ldv_kernel_resource_state != LDV_ALLOCATED)
        /* ASSERT Module can release only allocated kernel resource */
        ldv_assert();

    ldv_free(r);

    /* NOTE Release kernel resource */
    ldv_kernel_resource_state = LDV_NONALLOCATED;

}

void ldv_check_final_state(void)
{
```

(continues on next page)

(continued from previous page)

```
    if (ldv_kernel_resource_state != LDV_NONALLOCATED)
        /* ASSERT Kernel resource should be released at the end of work */
        ldv_assert();
}
```

Besides, it implicitly refers aspect *$KLEVER_SRC*/presets/jobs/specifications/simpleos/kernel/resource.aspect:

```
before: file("$this")
{
#include <kernel.h>

extern struct resource *ldv_allocate_resource(int);
extern void ldv_release_resource(struct resource *);
}

around: call(struct resource *allocate_resource(int init))
{
    return ldv_allocate_resource(init);
}

around: call(void release_resource(struct resource *r))
{
    ldv_release_resource(r);
}
```

The following requirements are checked by the given requirements specification:

- Each module can allocate kernel resource only once.
- Module can release only allocated kernel resource.
- Kernel resource should be released at the end of work.

memory safety checks for memory safety, i.e. absence of NULL pointer dereferences, buffer overflows, memory leaks, etc. Regarding performed checks it is similar to the namesake requirements specification dedicated for the Linux kernel, but it refers another model *$KLEVER_SRC*/presets/jobs/specifications/simpleos/memory safety/memory.c:

```
#include <kernel.h>
#include <ldv/verifier/memory.h>

struct resource *ldv_allocate_resource(int init)
{
    struct resource *r;

    if (init % 2)
        return NULL;

    r = ldv_xmalloc(sizeof(*r));
    r->x = init;

    return r;
}
```

(continues on next page)

(continued from previous page)

```

void ldv_release_resource(struct resource *r)
{
    ldv_free(r);
}
```

and aspect *$KLEVER_SRC*/presets/jobs/specifications/simpleos/memory safety/memory.aspect:

```
before: file("$this")
{
#include <kernel.h>

extern struct resource *ldv_allocate_resource(int);
extern void ldv_release_resource(struct resource *);
}

around: call(struct resource *allocate_resource(int init))
{
    return ldv_allocate_resource(init);
}

around: call(void release_resource(struct resource *r))
{
    ldv_release_resource(r);
}
```

1.9.3 Running Verification

After you described the new preset verification job, you need to populate related files once. For this you should open *Manager Tools* in the Klever web interface, select *preset jobs* and populate them.

Then you can create any number of verification jobs on the base of the populated preset verification job and start their decision. Corresponding configuration files, models, and specifications will be automatically updated from the repository each time if you are using Klever in the development mode. This process as well as expert assessment of verification results are considered in details in *Tutorial*.

After all if everything will go well, you will get 15 verification tasks (5 modules are verified against 3 requirement specifications) resulting to 4 unsafes and 11 safes:

- Requirements specification *empty* demonstrates only safes and code coverage as expected.
- Requirements specification *kernel:resource* catches 3 bugs corresponding to all its checks.
- Requirements specification *memory safety* reports a NULL pointer dereference.

1.9.4 Further Steps

You can try to adapt this tutorial to support verification of your program. Do not worry, if you will make any mistakes in file names and their contents along the way. Klever should report you rather clear error messages in this case, so that you will be able to fix different issues step by step. For instance, there was about 20 unsuccessful attempts at verification of Simple OS when developing everything necessary for this tutorial.

You should keep in mind that this tutorial does not cover all aspects of using Klever. In particular, it does not touch complicated environment modeling, development of advanced requirement specifications and some other things. Some of these topics are considered in separate sections of the given user documentation, e.g. you can find details for the former in *Development of Environment Model Specifications*.

If you will encounter any issues, and you will have any questions, please, contact us for support.

1.10 Developer Documentation

1.10.1 How to Write This Documentation

This documentation is created using Sphinx from reStructuredText source files. To improve existing documentation or to develop the new one you need to read at least the following chapters of the Sphinx documentation:

1. Defining document structure.
2. Adding content.
3. Running the build.
4. reStructuredText Primer.
5. Sphinx Markup Constructs.
6. Sphinx Domains (you can omit language specific domains).

Please, follow these advises:

1. Do not think that other developers and especially users are so smart as you are.
2. Clarify ambiguous things and describe all the details without missing anything.
3. Avoid and fix misprints.
4. Write each sentence on a separate line.
5. Do not use blank lines except it is required.
6. Write a new line at the end of each source file.
7. Break sentences longer than 120 symbols to several lines if possible.

To develop documentation it is recommended to use some visual editor.

> **Warning:** Please do not reinvent the wheel! If you are a newbie then examine carefully the existing documentation and create the new one on that basis. Just if you are a guru then you can suggest to improve the existing documentation.

1.10.2 Using Git Repository

Klever source code resides in the Git repository. There is plenty of very good documentation about Git usage. This section describes just rules specific for the given project.

Update

1. Periodically synchronize your local repository with the main development repository (it is available just internally at ISP RAS):

   ```
   branch $ git fetch origin
   branch $ git remote prune origin
   ```

   Note: This is especially required when you are going to create a new branch or to merge some branch to the master branch.

2. Pull changes if so:

   ```
   branch $ git pull --rebase origin branch
   ```

   Warning: Forget about pulling without rebasing!

3. Resolve conflicts if so.

Fixing Bugs and Implementing New Features

1. One must create a new branch to fix each individual bug or implement a new feature:

   ```
   master $ git checkout -b fix-conf
   ```

   Warning: Do not intermix fixes and implementation of completely different bugs and features into one branch. Otherwise other developers will need to wait or to make some tricky things like cherry-picking and merging of non-master branches. Eventually this can lead to very unpleasant consequences, e.g. the master branch can be broken because of one will merge there a branch based on another non working branch.

2. Push all new branches to the main development repository. As well re-push them at least one time a day if you make some commits:

   ```
   fix-conf $ git push origin fix-conf
   ```

3. Merge the master branch into your new branches if you need some recent bug fixes or features:

   ```
   fix-conf $ git merge master
   ```

   Note: Do not forget to update the master branch from the main development repository.

Note: Do not merge remote-tracking branches.

4. Ask senior developers to review and to merge branches to the master branch when corresponding bugs/features are fixed/implemented.
5. Delete merged branches:

```
master $ git branch -d fix-conf
```

1.10.3 Releases

Generally we follow the same rules as for development of the Linux kernel.

Each several months a new release will be issued, e.g. 0.1, 0.2, 1.0.

Just after this a merge window of several weeks will be opened. During the merge window features implemented after a previous merge window or during the given one will be merged to master.

After the merge window just bug fixes can be merged to the master branch. During this period we can issue several release candidates, e.g. 1.0-rc1, 1.0-rc2. Just before creating a new release it is necessary to bump up the Klever version in `docs/conf.py` and `setup.py` appropriately.

New releases should be described in `CHANGELOG.md` according to the used format. In addition, after issuing a new release we can decide to support a stable branch. This branch will start from a commit corresponding to the given release. It can contain just bug fixes relevant to an existing functionality and not to a new one which is supported within a corresponding merge window.

1.10.4 Updating List of Required Python Packages

To update the list of required Python packages first you need to install Klever package from scratch in the newly created virtual environment without using the old *requirements.txt* file. Run the following commands within *$KLEVER_SRC*:

```
$ python3 -m venv venv
$ source venv/bin/activate
$ pip install --upgrade pip wheel setuptools setuptools_scm
$ pip install .
```

This will install latest versions of required packages. After confirming that Klever works as expected, you should run the following command within *$KLEVER_SRC*:

```
$ python -m pip freeze | grep -v 'klever' > requirements.txt
```

Updated list of requirements will be saved and should be committed to the repository afterwards.

Besides, you should also update the list of required Python packages for OpenStack deployment. For this purpose you need to run the following commands after executing commands above:

```
$ pip install ".[openstack]"
$ python -m pip freeze | grep -v 'klever' > requirements-openstack.txt
```

At last you should specify actual versions of Python packages in `docs/requirements.txt`. Note, that Sphinx 4.4.0 requires docutils < 0.18.

1.10.5 How to generate build bases for testing Klever

Most likely you can get actual, prepared in advance build bases for testing Klever from *ldvuser@ldvdev:build-bases/build-bases.tar.xz* (this works just within the ISP RAS local network).

To (re)generate build bases for testing Klever you need to do as follows:

1. Make actions described in *Providing appropriate compilers*. After all there should be directory `build bases/gcc48` with GCC 4.8 binaries within *$KLEVER_SRC*. Symbolic links are not accepted.
2. Make actions described in *Providing sources*. After all within *$KLEVER_SRC* there should be directories `build bases/linux-stable` and `build bases/busybox` with Git repositories of the Linux kernel stable and BusyBox respectively. Symbolic links are not accepted.
3. Execute following commands within *$KLEVER_SRC* (you may need to run them using *sudo*, the first command can take several hours depending on your hardware):

```
$ docker build -t build-bases -f Dockerfile.build-bases .
$ docker create --name dummy build-bases
$ docker cp dummy:/usr/src/build-bases.tar.xz build\ bases/
$ docker rm dummy
```

4. Periodically run following commands to clean up useless containers and images, especially after failed builds (this can help to considerably reduce the occupied disk space):

```
$ docker ps --filter status=exited -q | xargs docker rm
$ docker images -q -f dangling=true | xargs docker rmi
```

After that the archive with generated build bases will be located at `build bases/build-bases.tar.xz`.

Besides, you can follow the following steps:

1. Install Klever locally for development purposes according to the user documentation (see *Deployment*).
2. Create a dedicated directory for sources and build bases and move to it. Note that there should be quite much free space. We recommend at least 100 GB. In addition, it would be best of all if you will name this directory “build bases” and create it within the root of the Klever Git repository (this directory is not tracked by the repository).
3. Make actions described in *Providing sources*.
4. Make actions described in *Providing appropriate compilers*.
5. Run the following command to find out available descriptions of build bases for testing Klever:

```
$ klever-build -l
```

6. Select appropriate build bases descriptions and run the command like below:

```
$ klever-build "linux/testing/requirement specifications" "linux/testing/common␣
↪models"
```

7. Wait for a while. Prepared build bases will be available within directory “build bases”. Note that there will be additional identifiers, e.g. “build bases/linux/testing/6e6e1c”. These identifiers are already specified within corresponding preset verification jobs.
8. You can install prepared build bases using deployment scripts, but it is boring. If you did not follow an advice regarding the name and the place of the dedicated directory from item 2, you can create a symbolic link with name “build bases” that points to the dedicated directory within the root of the Klever Git repository.

Providing appropriate compilers

Most of build bases for testing Klever could be built using GCC 4.8 on Debian or Ubuntu. Otherwise there is an explicit division of build bases descriptions, e.g.:

- linux/testing/environment model specifications/gcc48
- linux/testing/environment model specifications/gcc63

(the former requires GCC 4.8 while the latter needs GCC 6.3 at least).

That's why you may need to get GCC 4.8 and make it available through PATH. Users of some other Linux distributions, e.g. openSUSE 15.1, can leverage the default compiler for building all build bases for testing Klever.

The simplest way to get GCC 4.8 on Ubuntu is to execute the following commands:

```
$ sudo apt update
$ sudo apt install gcc-4.8
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 70
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 48
$ sudo update-alternatives --config gcc
```

(after executing the last command you need to select GCC 4.8; do not forget to make v.v. after preparing build bases!)

You can download prepared in advance GCC 4.8 binaries from here. They work on Debian 9. Besides, they can work on Ubuntu and new versions of Debian.

Providing sources

You should clone a Linux kernel stable Git repository to *linux-stable* (scripts prepare build bases for different versions of the Linux kernel for which the Git repository serves best of all), e.g.:

```
$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/ linux-
↪stable
```

You can use alternative sources of the Git repository, if the above one is not working well and fast enough:

1. https://kernel.googlesource.com/pub/scm/linux/kernel/git/stable/linux-stable
2. https://github.com/gregkh/linux

Also, you should clone a BusyBox Git repository, e.g.:

```
$ git clone git://busybox.net/busybox.git
```

1.10.6 Translating Web User Interface

Currently the Klever web UI is translated only to Russian. To update existing translations, you should run the following commands within *$KLEVER_SRC*:

```
$ cd bridge
$ ./manage.py makemessages --all --ignore logs --ignore media --ignore run --ignore␣
↪static --ignore tools/error-traces
```

Then you will need to carefully examine changed *.po* files and make necessary fixes. To (re)generate appropriate binary files to be used at showing translated messages, you should run the following commands within *$KLEVER_SRC*:

```
$ cd bridge
$ ./manage.py compilemessages
```

1.10.7 Generating Bare CPAchecker Benchmarks

Development of Klever and development of CPAchecker are not strongly coupled. Thus, verification tasks that are used for testing/validation of Klever including different versions and configurations of CPAchecker as back-ends may be useful to track regressions of new versions of CPAchecker. This should considerably simplify updating CPAchecker within Klever (this process usually involves a lot of various activities both in Klever and in CPAchecker; these activities can take enormous time to be completed that complicates and postpones updates considerably). In addition, this is yet another test suite for CPAchecker. In contrast to other test suites this one likely corresponds to the most industry close use cases.

One can (re-)generate bare CPAchecker benchmarks almost automatically. To do this it is recommended to follow next steps:

1. Clone https://gitlab.com/sosy-lab/software/ldv-klever-benchmarks.git or git@gitlab.com:sosy-lab/software/ldv-klever-benchmarks.git once.
2. After some changes within Klever specifications, configurations and test cases you need to solve appropriate verification jobs. To avoid some non-determinism it is better to use the same machine, e.g. LDV Dev, to do this. Though particular verification jobs to be solved depend on changes made, in ideal, it is much easier to consider all verification jobs at once to avoid any tricky interdependencies (even slight improvements or fixes of some specifications may result in dramatic and unexpected changes in some verification results).
3. Download archives with verifier input files for each solved verification jobs to the root directory of the cloned repository.
4. Run “python3 make-benchs.py” there.
5. Estimate changes in benchmarks and verification tasks (there is not any formal guidance). If you agree with these changes, then you need to commit them and to push to the remote. After that one may expect that new commits to trunk of the CPAchecker repository will be checked for regressions against an updated test suite.

1.10.8 Using PyCharm IDE

To use PyCharm IDE for developing Klever follow the following steps.

Installation

1. Download PyCharm Community from https://www.jetbrains.com/pycharm/download/ (below all settings are given for version 2018.8.8, you have to adapt them for your version by yourself).
2. Follow installation instructions provided at that site.

Setting Project

At the "Welcome to PyCharm" window:

1. Specify your preferences.
2. *Open*.
3. Specify the absolute path to directory *$KLEVER_SRC*.
4. *OK*.

Configuring the Python Interpreter

1. *File* → *Settings* → *Project: Klever* → *Project Interpreter* → *Settings* → *Show all…*.
2. Select the Python interpreter from the Klever Python virtual environment.
3. *OK*.
4. Select the added Python interpreter from the list and press `Enter`.
5. Input *Python 3.10 (klever)* in field *name*.
6. *OK*.
7. For the rest projects select *Python 3.10 (klever)* in field *Project Interpreter*.

Setting Run/Debug Configuration

Common run/debug configurations are included into the Klever project. Common configurations with names starting with **$** should be copied to configurations with names without **$** and adjusted in accordance with instructions below. If you want to adjust configurations with names that not starting with **$** you also have to copy them before.

1. *Run* → *Edit Configurations…*.

Klever Bridge Run/Debug Configuration

Note: This is available just for PyCharm Professional.

- Specify *0.0.0.0* in field *Host* if you want to share your Klever Bridge to the local network.
- Specify your preferred port in field *Port*.

Note: To make your Klever Bridge accessible from the local network you might need to set up your firewall accordingly.

Klever Core Run/Debug Configuration

This run/debug configuration is only useful if you are going to debug Klever Core.

- Extend existing value of environment variable `PATH` so that CIF (`cif` or `compiler`), Aspectator (`aspectator`) and CIL (`toplevel.opt`) binaries could be found (edit value of field *Environment variables*).
- Specify the absolute path to the working directory in field *Working directory*.

Note: Place Klever Core working directory somewhere outside the main development repository.

Note: Klever Core will search for its configuration file `core.json` in the specified working directory. Thus, the best workflow to debug Klever Core is to set its working directory to the one created previously when it was run without debugging. Besides, you can provide this file by passing its name as a first parameter to the script.

Documentation Run/Debug Configuration

Specify another representation of documentation in field *Command* if you need it.

Testing

Klever Bridge Testing

Note: This is available just for PyCharm Professional.

1. *Tools → Run manage.py Task…*:

```
manage.py@bridge > test
```

Note: To start tests from console:

```
$ cd bridge
$ python3 manage.py test
```

Note: Another way to start tests from console:

```
$ python3 path/to/klever/bridge/manage.py test bridge users jobs reports marks service
```

Note: The test database is created and deleted automatically. If the user will interrupt tests the test database will preserved and the user will be asked for its deletion for following testing. The user should be allowed to create databases (using command-line option *–keedb* does not help).

Note: PyCharm has reach abilities to analyse tests and their results.

Additional documentation

A lot of useful documentation for developing Django projects as well as for general using of the PyCharm IDE is available at the official site.

1.10.9 Using Visual Studio Code

Klever includes quite much C stuff such as models and tests. You may want to use VS Code to develop it since VS Code can find errors, suggest proper entity names and so on. First of all you need to install an appropriate extension for support of C/C++.

There is a basic configuration for VS Code in Klever already. It excludes some directories from search and indexing, refers Klever directories with auxiliary headers and so on. If you use headers (functions, types, macros, etc.) from your project within models and tests, you need to configure VS Code appropriately. This strongly depends on particular project and its build process. We suggest to refer headers that are placed to directory *Storage* of the Clade's build base since there will be only those ones that are suitable for those configuration and architecture of the target program, that you are going to verify. For instance, for common models, requirement specifications and their tests you can specify following directories to search for headers as a value of option *C_Cpp.default.includePath* for your local user's settings:

- abs_path_to_clade_build_base/Storage/src/linux-5.12-rc3/include/
- abs_path_to_clade_build_base/Storage/src/linux-5.12-rc3/arch/x86/include/
- abs_path_to_clade_build_base/Storage/src/linux-5.12-rc3/arch/x86/include/uapi/
- abs_path_to_clade_build_base/Storage/src/linux-5.12-rc3/arch/x86/include/generated/

Moreover, to resolve some conflicts you may have to set "CONFIG_SMP" as a value of option *C_Cpp.default.defines*. Please, do not include project-specific settings to the common settings of VS Code stored within directory *.vscode*.

1.10.10 Extended Violation Witness Format

The original format of violation witnesses is intended primarily for automatic validation. Each violation witness can describe a subset of possible execution paths and lack some important details. This hinders their manual analysis by experts.

We suggest the extended format of violation witnesses to enhance their visualization and assessment capabilities. This format requires an extended violation witness to represent a single error path as accurate as possible, i.e. it should refer all expressions, statements and declarations starting from an entry point and up to a found violation as well as all global variable declarations. Besides, extended violation witnesses should mandatory use *enterFunction* and *returnFromFunction* tags for all functions that are called along the error path and have definitions.

To distinguish declarations from statements and expressions, especially, to separate global variable declarations from the entry point, we suggest to introduce an additional data tag *declaration*. Its value should be *true* for all edges corresponding to global and local declarations. Its default value used for all other edges implicitly should be *false*.

One more extension is intended for adding important internal information from verification tools to violation witnesses. For instance, when checking memory safety verification tools can point out places where leaked memory is allocated. The corresponding data tag is *note*. Its value should has the following format:

```
level="N" hide="true|false" value="Some meaningful text"
```

N sets the importance of the note. It should be in range from 0 to 3 where 0 should be used just for edges corresponding to found violations. Level 1 should be used for vital notes since these notes will be shown by default and they will be used for obtaining *error trace patterns* used for automatic assessment of similar violation witnesses. All levels of notes will be specially highlighted at visualization. Attribute *hide* controls whether notes should be shown together with corresponding edges (in case when *hide* is *false*) or without it (otherwise). Edges can be omitted when notes represent enough information about them in their attribute *value*. The example of this data tag value is as follows:

```
level="0" hide="false" value="Memory leak of calloc_ID13 is detected"
```

Verification tools can provide multiple *note* data tags per an edge.

Thus, the extended format of violation witnesses does extend the existing format of violation witnesses. Extended violation witnesses can be even validated like non-extended ones.

1.10.11 Error Trace Format

We suggest converting violation witnesses in the extended format represented above to error traces that are more convenient for visualization and assessment purposes. Error traces should be represented as JSON files with the following content:

```
{
    "format": 1,
    "files": [
        "filename1",
        "filename2",
        "..."
    ],
    "global variable declarations": [
        {
            "file": 0,
            "line": 1,
            "source": "struct module x;"
        },
        {
            "file": 0,
            "line": 2,
            "source": "static ldv_counter = 1;",
            "notes": [
                {
                    "level": 1
                    "text": "Initialize counter to zero"
                }
            ],
            "hide": true
        },
        {
        }
    ],
    "trace": "NodeObject"
}
```

format indicates a current version of the error trace format. For all changes in syntax and especially semantics of the represented data it should be changed.

files lists all filenames referred by the error trace. Below particular files are represented as indexes in this array. This is necessary for optimization purposes since there may be very many edges corresponding to different files that can have rather long paths.

For global variable declarations *file*, *line* and *source* are mandatory attributes. Their meaning is quite obvious. *notes* and *hide* correspond to entities from the extended violation witnesses straightforwardly. Below we present a bit more details on these attributes.

NodeObject represents the error path (error trace) starting from the entry point and finishing at the detected violation. It is a JSON object with following attributes:

- *type* - one of “thread”, “action”, “declarations”, “declaration”, “statement” and “function call”.
- *thread* - a thread identifier. This attribute is mandatory for objects of type “thread”.
- *file* - an index in the array of files presented above. This attribute is mandatory for objects of types “action”, “declaration”, “statement” and “function call”.
- *line* - a line number in this file. This attribute is mandatory for the same objects as *file*.
- *source* - a piece of the source code corresponding to a violation witness edge. This attribute is mandatory for objects of types “declaration”, “statement” and “function call”.
- *highlight* - highlighting for a given piece of the source code. This attribute can be set for the same objects as *source*. Its value is an array of arrays each containing a highlight class that influences visualization, a start offset and an end offset of a corresponding entity. All offsets should be in a *source* length range, they should not overlap and the end offset should be greater than the start offset.
- *condition* - either true or false depending on a corresponding edge represents a conditional statement or not respectively. This attribute can be sef for objects of types “statement” and “function call”.
- *assumption* - verification tool assumptions coinciding with a value of *assumption* data tag. This attribute can be sef for objects of types “statement” and “function call”.
- *display* - a text replacing *source*, e.g. instead of a complete function call statement just a function name can be shown if it is stored as a value of this attribute. This attribute is mandatory for objects of types “action” and “function call”. Also, it can be set for objects of types “declaration” and “statement”.
- *relevant* - either true or false that denotes actions that are relevant and irrelevant for creating error trace patterns. This attribute is mandatory for objects of type “action”. By default its value is false.
- *notes* - a list of notes like demonstrated above. This attribute is mandatory for objects of types “declaration”, “statement” and “function call”.
- *hide* - either true of false that correspondingly hides or shows a corresponding *source* or *display*. This attribute is mandatory for the same objects as *notes*. By default its value is false.
- *children* - a list of elements each of type *NodeObject*. This attribute is mandatory for objects of types “thread”, “action”, “declarations” and “function call”.

The first *NodeObject* should have the *thread* type.

1.10.12 Code Coverage Format

We suggest to convert code coverage reports from verification tools to the more appropriate form for their visualization. Converted code coverage reports should be represented as JSON files. There are should be JSON files for all source files that were covered somehow as well as one file per a verification task with statistics. Code coverage for individual source files should be placed to files *path/to/src_file.cov.json* and they should have the following content:

```
{
    "format": 1,
    "line coverage": {
        "1": 4,
        "3": 7,
        "...": "..."
    },
    "function coverage": {
        "1": 1,
        "17": 0,
        "...": "..."
    },
    "notes": {
        "19": {
            "kind": "Verifier assumption",
            "text": "Inline Assembler is ignored"
        },
        "51": {
            "kind": "Environment modelling hint",
            "text": "Function \"driver_release\" may be called within context of \
↪"driver_probe\" and \"driver_disconnect\" entry points"
        },
        "...": "..."
    }
}
```

format means the same as the error trace format considered above.

line coverage and *function coverage* shows the number of states for corresponding lines of code. For functions these lines of code coincide with places where they are defined. The number of states reflect time spent for verification of lines and functions to some extent.

notes enumerate hints from verification tools or Klever itself for corresponding lines of code. Each such hint can have a random text and one of predefined kinds. For each kind a dedicated style will be used at visualization.

Code coverage statistics should be put to file *coverage.json* of the following content:

```
{
    "format": 1,
    "coverage statistics": {
        "path/to/src": [100, 1000, 5, 10],
        "...": []
    },
    "most covered lines": [
          "path/to/src:333",
          "path/to/another/src:33",
          "path/to/src:233",
          "..."
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

format means the same as the error trace format considered above.

coverage statistics represents the number of covered lines, the number of lines that could be covered potentially, the number of covered functions and the number of functions that could be covered potentially for corresponding source files.

most covered lines enumerates source files and lines within them that were covered most times.

The same format is appropriate for representing code coverage for the whole program independently for each requirements specification that is also supported by Klever.

1.11 Glossary

Environment model
Environment models emulate interactions of target programs or *program fragments* like Linux kernel loadable modules with their environment like libraries, user inputs, interruptions and so on. Ideally they should cover only those interaction scenarios that are possible during real executions, but usually this is not the case, so false alarms and missing bugs take place. Klever generates each environment model on the basis of specifications and it is represented as a number of additional C source files (*models*) bound with original ones through instrumentation.

Program fragment
Software verification tools can not check nontrivial requirements for large programs in a reasonable time. So Klever decomposes target programs into moderate-sized program fragments and generates independent verification tasks for all of them. Each program fragment represents a subset of program source files and information how to build them. Program fragments can intersect with each other. Linux kernel loadable modules can serve as a good example of program fragments.

$KLEVER_SRC
A path to a root directory of a Klever source tree.

$KLEVER_DEPLOY_DIR
A path to a directory where Klever should be deployed. Although this directory can be one of standard ones like `/usr/local/bin` or `/bin`, it is recommended to use some specific one.

$SSH_RSA_PRIVATE_KEY_FILE
A path to a file with SSH RSA private key. It is not recommended to use your sensitive keys. Instead either create and use a specific one or use keys that are accepted in your groups to enable an access to other group members.

$OS_USERNAME
Username used to login to OpenStack.

$INSTANCES
A number of OpenStack instances to be deployed.

BIBLIOGRAPHY

[G20] Gratinskiy V.A., Novikov E.M., Zakharov I.S. Expert Assessment of Verification Tool Results. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), volume 32, issue 5, pp. 7-20. 2020. https://doi.org/10.15514/ISPRAS-2020-32(5)-1. (In Russian)

[N13] E.M. Novikov. An approach to implementation of aspect-oriented programming for C. Programming and Computer Software, volume 39, issue 4, pp. 194-206, Springer US, 2013. https://doi.org/10.1134/S0361768813040051

[N13-2] E.M. Novikov. Building Programming Interface Specifications in the Open System of Componentwise Verification of the Linux Kernel. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), volume 24, pp. 293-316, 2013 (In Russian). https://doi.org/10.15514/ISPRAS-2013-24-13

[Z18] I. Zakharov, E. Novikov. Compositional Environment Modelling for Verification of GNU C Programs. In Proceedings of the 2018 Ivannikov Ispras Open Conference (ISPRAS'18), pp. 39-44, IEEE Computer Society, 2018, https://doi.org/10.1109/ISPRAS.2018.00013.

[N18] E. Novikov, I. Zakharov. Verification of Operating System Monolithic Kernels Without Extensions. In: Margaria T., Steffen B. (eds) Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation. Industrial Practice (ISoLA'18), LNCS, volume 11247, pp. 230–248, Springer, Cham, 2018. https://doi.org/10.1007/978-3-030-03427-6_19.

[ZN18] E. Novikov, I. Zakharov. Towards automated static verification of GNU C programs. In: Petrenko A., Voronkov A. (eds) Proceedings of the 11th International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI'17), LNCS, volume 10742, pp. 402–416, Cham, Springer, 2018. https://doi.org/10.1007/978-3-319-74313-4_30.

INDEX